Deep Learning With Python

Develop Deep Learning Models on Theano and TensorFlow Using Keras

Jason Brownlee

MACHINE
LEARNING
MASTERY



目錄

深度学习：Python 教程 (Deep Learning With Python)

Deep Learning With Python: Develop Deep Learning Models on Theano and TensorFlow Using Keras

使用 Keras、Python、Theano 和 TensorFlow 开发深度学习模型

原书网站：https://machinelearningmastery.com/deep-learning-with-python/

作者：Jason Brownlee

译者：cnbeining

来源：cnbeining/deep-learning-with-python-cn

前言

深度学习一直很有意思：虽然神经网络不是新鲜事，但是直到最近才开始迅速发展。得益于硬件、技术和开源软件的升级，现在创建庞大的神经网络已经无比容易。

随着神经网络变得更大更深，机器学习可以解决很多问题。我个人近几年的观察是，神经网络在很多领域的效果是惊艳的，包括但不限于物体识别、语音识别、分类、机器翻译等等等等等等等等。

那么问题来了：深度学习应该从哪里入门？鉴于目前机器学习没有好的用Python的入门书，我撰写了本书。

本书选用Keras：因为Keras简单明了，封装了所有的底层细节，只留下最重要的API，可以很快捷地开始神经网络的开发。

我希望我当初有这样一本入门书：最后祝您调参愉快！

Jason Brownlee

写于澳大利亚墨尔本

2016

译者的话

很有幸翻译本书。

本书的阅读和翻译一气呵成。我尽可能言简意赅的翻译，以便和自己的稿费（万一有呢）过不去。

本书的风格类似fast.ai。然而fast.ai并不使用Keras（否则这个译本就不会出现了）：如果想用Keras，这里有示例代码。

如果您没有Python基础，请阅读廖雪峰的Python学习教程。

本书的Theano部分翻译并不认眞，因为Theano的死期已定：请使用TensorFlow作为后端吧。

希望本书可以填补Keras资料的空白。

本书的翻译得到了很多指正：没有他们本书的准确性将大打折扣。如果我忘记了您的名字，请联系我：i@cnbeining.com 在此致谢。

（名称不分先后）

msg7086、TsukaTsuki、CAMOE

Beining

第一部分 入门

第一章 深度学习入门

欢迎购买本书：本书旨在帮助您使用Python进行深度学习，包括如何使用Keras构建和运行深度学习模型。本书也包括深度学习的技巧、示例代码和技术内容。

深度学习的数学基础很精妙：但是一般用户不需要完全了解数学细节就可以抄起键盘开始编程。实用一点讲，深度学习并不复杂，带来的成效却很客观。教会你如何用深度学习：这就是本书的目的。

1.1 深度学习：如何错误入门

如果你去问大佬们深度学习如何入门，他们会怎么说？不外乎：

- 线性代数是关键啊！
- 你得了解传统神经网络才能干啊！
- 概率论和统计学是基础的基础不是吗？
- 你得先在机器学习的水里扑腾几年再来啊。
- 不是计算机博士不要和我说话好吗！
- 入门挺简单的：10年经验应该差不多也行有可能就够了吧。

总结一下：只有大神才能做深度学习。

净**TM**扯淡！

1.2 使用Python进行深度学习

本书准备把传统的教学方式倒过来：直接教你怎么深度学习。如果你觉得这东西真厉害我要好好研究一下，再去研究理论细节。本书直接让你用深度学习写出能跑的东西。

我用了不少深度学习的库：我觉得最好的还是基于Python的Keras。Python是完整的成熟语言，可以直接用于商业项目的核心，这点R是比不上的。和Java比，Python有SciPy和scikit-learn这些专业级别的包，可以快速搭建平台。

Python的深度学习库有很多，最著名的是蒙特利尔大学的Theano（已死，有事烧纸）和Google的TensorFlow。这两个库都很简单，Keras都无缝支持。Keras把数值计算的部分封装掉，留下搭建神经网络和深度学习模型的重点API。

本书会带领你亲手构建神经网络和深度学习模型，告诉你如何在自己的项目中利用。废话少说，赶快开始：

1.3 本书结构

本书分3部分：

- 课程：介绍某个神经网络的某个功能，以及如何使用Keras的API写出来
- 项目：将课上的知识放在一起，写一个项目：这个项目可以作为模板
- 示例：直接可以复制粘贴的代码！本书还附赠了很多代码，在Github上！

1.3.1 第一部分：课程和项目

每节课是独立的，推荐一次性完成，时长短则20分钟，长则数小时 - 如果你想仔细调参数。课程分4块：

- 背景
- 多层感知器
- 高级多层感知器和Keras
- 卷积神经网络

1.3.2 第二部分：背景知识

这部分我们介绍Theano、TensorFlow（TF）和Keras这3个库，以及如何在亚马逊的云服务（AWS）上用低廉的价格测试你的网络。分成4个部分：

- Theano 入门
- TensorFlow 入门
- Keras 入门

这些是最重要的深度学习库。我们多介绍一点东西：

- 项目：在云上部署GPU项目

到这里你应该准备好用Keras开发模型了。

1.3.3 第三部分：多层感知器

这部分我们介绍前馈神经网络，以及如何用Keras写出自己的网络。大体分段：

- 多层感知器入门
- 用Keras开发第一个神经网络
- 测试神经网络模型性能
- 用Scikit-Learn和Keras模型进行机器学习

这里有3个项目可以帮助你开发神经网络，以及为之后的网络打下模板：

- 项目：多类分类
- 项目：分类问题
- 项目：回归问题

到这里你已经熟悉了Keras的基本操作。

1.3.4 第四部分：高级多层感知器

这部分我们进一步探索Keras的API，研究如何得到世界顶级的结果。内容包括：

- 如何保存神经网络
- 如何保存最好的网络
- 如何边训练观察训练结果
- 如何对付过拟合
- 如何提高训练速度

到这里你已经可以使用Keras开发成熟的模型了。

1.3.5 第五部分：卷积神经网络（CNN）

这部分我们介绍一些计算机视觉和自然语言的问题，以及如何用Keras构建神经网络出色地解决问题。内容包括：

- 卷积神经网络入门
- 如何增强模型效果

写代码才能真正理解网络：这里我们用CNN解决如下问题：

- 项目：手写字符识别
- 项目：图像物体识别
- 项目：影视评论分类

到这里你可以用CNN对付你遇到的实际问题了。

1.3.6 结论

这部分我们给你提供一些继续深造的资料。

1.3.7 示例

边学习边积累代码库：每个问题你都写了代码，供以后使用。

本书给你所有项目的代码，以及一些没有讲到的Keras代码。自己动手积累吧！

1.4 本书需求

1.4.1 Python和SciPy

你起码得会装Python和SciPy，本书默认你都配置好了。你可以在自己的机器上，或者虚拟机/Docker/云端配置好环境。参见第二章项目。

本书使用的软件和库：

- Python 2或3：本书用版本2.7.11.
- SciPy和NumPy：本书用SciPy 0.17.0和NumPy 1.11.0.
- Matplotlib：本书用版本1.5.1
- Pandas：本书用版本0.18.0
- scikit-learn：本书用版本0.17.1。

版本不需要完全一致：但是希望安装的版本不要低于上面的要求。第二部分会带领你配置环境。

1.4.2 机器学习

你不需要专业背景，但是会用scikit-learn研究简单的机器学习很有帮助。交叉检验等基本概念了解一下。书后有参考资料：简单阅读一下。

1.4.3 深度学习

你不需要知道算法的数学理论，但是概念需要有所了解。本书有个神经网络和模型的入门，但是不会深度研究细节。术后有参考资料：希望你对神经网络有点概念。

注意：所有的例子都可以用CPU跑，GPU不是必备的，但是GPU可以显著加速运算。第5章会告诉你如何在云上配置GPU。

1.5 本书目标

希望你看完本书后有能力从数据集上用Python开发深度学习算法。包括：

- 如何开发并测试深度学习模型
- 如何使用高级技巧
- 如何为图片和文本数据构建大模型
- 如何扩大图片数据
- 如何寻求帮助

现在可以开始了。你可以挑自己需要的主题阅读，也可以从头到尾走一遍流程。我推荐后者。

希望你亲手做每一个例子，将所思所想记录下来。我的邮箱是jason@MachineLearningMastery.com。本书希望你努力一下，尽快成为深度学习工程师。

1.6 本书不是什么

本书为开发者提供深度学习的入门教程，但是挂一漏万。本书不是：

- 深度学习教科书：本书不深入神经网络的理论细节，请自行学习。
- 算法书：我们不关注算法如何工作，请自行学习。
- Python编程书：本书不深入讲解Python的用法，希望你已经会Python了。

如果需要深入了解某个主题，请看书后的帮助。

1.7 总结

此时此刻，深度学习的工具处于历史顶峰，神经网络和深度学习的发展从未如此之快，在无数领域出神入化。希望你玩的开心。

1.7.1 下一步

下一章我们讲解一下Theano、TensorFlow和你要用的Keras。

第二部分 背景

略

第3章 TensorFlow入门

TensorFlow是Google创造的数值运算库，作为深度学习的底层使用。本章包括：

- TensorFlow介绍
- 如何用TensorFlow定义、编译并运算表达式
- 如何寻求帮助

注意：TensorFlow暂时不支持Windows，你可以用Docker或虚拟机。Windows用户可以不看这章。

3.1 TensorFlow是什么？

TensorFlow是开源数学计算引擎，由Google创造，用Apache 2.0协议发布。TF的API是Python的，但底层是C++。和Theano不同，TF兼顾了工业和研究，在RankBrain、DeepDream等项目中使用。TF可以在单个CPU或GPU，移动设备以及大规模分布式系统中使用。

3.2 安装TensorFlow

TF支持Python 2.7和3.3以上。安装很简单：

```
sudo pip install TensorFlow
```

就好了。

3.3 TensorFlow例子

TF的计算是用图表示的：

- 节点：节点进行计算，有一个或者多个输入输出。节点间的数据叫张量：多维实数数组。
- 边缘：定义数据、分支、循环和覆盖的图，也可以进行高级操作，例如等待某个计算完成。
- 操作：取一个输入值，得出一个输出值，例如，加减乘除。

3.4 简单的TensorFlow

简单说一下TensorFlow：我们定义a和b两个浮点变量，定义一个表达式（c=a+b），将表达式变成函数，编译，进行计算：

```
import tensorflow as tf
# declare two symbolic floating-point scalars
a = tf.placeholder(tf.float32)
b = tf.placeholder(tf.float32)
# create a simple symbolic expression using the add function add
 = tf.add(a, b)
# bind 1.5 to 'a', 2.5 to 'b', and evaluate 'c'
sess = tf.Session()
binding = {a: 1.5, b: 2.5}
c = sess.run(add, feed_dict=binding)
print(c)
```

结果是4： 1.5+2.5=4.0。大的矩阵操作类似。

3.5 其他深度学习模型

TensorFlow自带很多模型，可以直接调用：首先，看看TensorFlow的安装位置：

```
python -c 'import os; import inspect; import tensorflow; print(o
s.path.dirname(inspect.getfile(tensorflow)))'
```

结果类似于：

```
/usr/lib/python2.7/site-packages/tensorflow
```

进入该目录，可以看见很多例子：

- 多线程word2vec mini-batch Skip-Gram模型
- 多线程word2vec Skip-Gram模型
- CIFAR-10的CNN模型
- 类似LeNet-5的端到端的MNIST模型
- 带注意力机制的端到端模型

example目录带有MNIST数据集的例子，TensorFlow的网站也很有帮助，包括不同的网络、数据集。TensorFlow也有个网页版，可以直接试验。

3.6 总结

本章关于TensorFlow。总结一下：

- TensorFlow和Theano一样，都是数值计算库
- TensorFlow和Theano一样可以直接开发模型
- TensorFlow比Theano包装的好一些

3.6.1 下一章

下一章我们研究Keras：我们用这个库开发深度学习模型。

第4章 Keras入门

Python的科学计算包主要是Theano和TensorFlow：很强大，但有点难用。Keras可以基于这两种包之一方便地建立神经网络。本章包括：

- 使用Keras进行深度学习
- 如何配置Keras的后端
- Keras的常见操作

我们开始吧。

4.1 Keras是什么？

Keras可以基于Theano或TensorFlow建立深度学习模型，方便研究和开发。Keras可以在Python 2.7或3.5运行，无痛调用后端的CPU或GPU网络。Keras由Google的Francois Chollet开发，遵循以下原则：

- 模块化：每个模块都是单独的流程或图，深度学习的所有问题都可以通过组装模块解决
- 简单化：提供解决问题的最简单办法，不加装饰，最大化可读性
- 扩展性：新模块的添加特别容易，方便试验新想法
- Python：不使用任何自创格式，只使用原生Python

4.2 安装Keras

Keras很好安装，但是你需要至少安装Theano或TensorFlow之一。

使用PyPI安装Keras：

```
sudo pip install keras
```

本书完成时，Keras的最新版本是1.0.1。下面这句话可以看Keras的版本：

```
python -c "import keras; print keras.__version__"
```

Python会显示Keras的版本号，例如：

```
1.0.1
```

Keras的升级也是一句话：

```
sudo pip install --upgrade keras
```

4.3 配置Keras的后端

Keras是Theano和TensorFlow的轻量级API，所以必须配合后端使用。后端配置只需要一个文件：

```
~/.keras/keras.json
```

里面是：

```
{"epsilon": 1e-07, "floatx": "float32", "backend": "theano"}
```

默认的后端是`theano`，可以改成`tensorflow`。下面这行命令会显示Keras的后端：

```
python -c "from keras import backend; print backend._BACKEND"
```

默认会显示：

```
Using Theano backend.
theano
```

变量`KERAS_BACKEND`可以控制Keras的后端，例如：

```
KERAS_BACKEND=tensorflow python -c "from keras import backend; p
rint backend._BACKEND"
```

会输出：

```
Using TensorFlow backend.
tensorflow
```

4.4 使用Keras搭建深度学习模型

Keras的目标就是搭建模型。最主要的模型是`Sequential`：不同层的叠加。模型创建后可以编译，调用后端进行优化，可以指定损失函数和优化算法。

编译后的模型需要导入数据：可以一批批加入数据，也可以一次性全加入。所有的计算在这步进行。训练后的模型就可以做预测或分类了。大体上的步骤是：

1. 定义模型：创建`Sequential`模型，加入每一层
2. 编译模型：指定损失函数和优化算法，使用模型的`compile()`方法
3. 拟合数据：使用模型的`fit()`方法拟合数据
4. 进行预测：使用模型的`evaluate()`或`predict()`方法进行预测

4.5 总结

本章关于Keras。总结一下：

- Keras是Theano和TensorFlow的封装，降低了复杂性
- Keras是最小化、模块化的封装，可以迅速上手
- Keras可以通过定义-编译-拟合搭建模型，进行预测

4.5.1 下一章

这是Python机器学习的最前沿：下个项目我们一步步在云上搭建机器学习的环境。

TODO

第三部分 多层感知器

第6章 多层感知器入门

神经网络很神奇，但是一开始学起来很痛苦，涉及大量术语和算法。本章主要介绍多层感知器的术语和使用方法。本章将：

- 介绍神经网络的神经元、权重和激活函数
- 如何使用构建块建立网络
- 如何训练网络

我们开始吧。

6.1 绪论

本节课的内容很多：

- 多层感知器
- 神经元、权重和激活函数
- 神经元网络
- 网络训练

我们从多层感知器开始谈起。

6.2 多层感知器

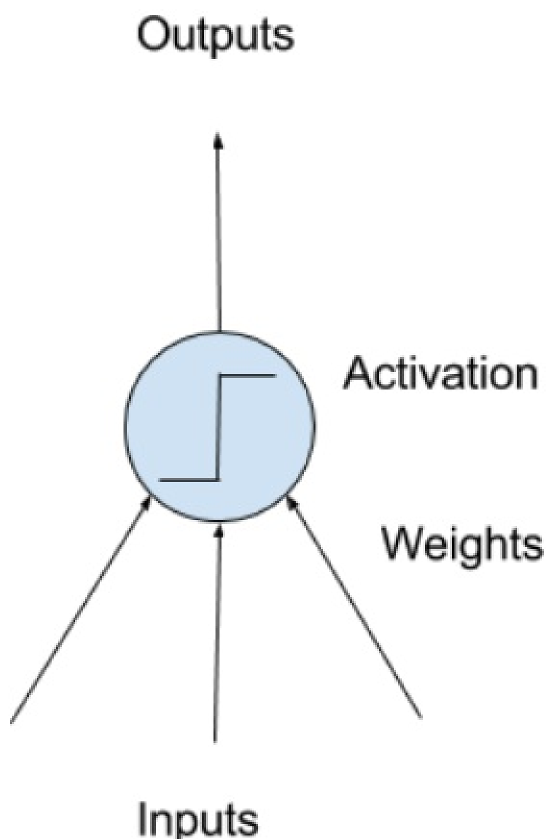
（译者注：本书中的“神经网络”一般指“人工神经网络”）

在一般的语境中，`人工神经网络`一般指`神经网络`，或者，`多层感知器`。感知器是简单的神经元模型，大型神经网络的前体。这个领域主要研究大脑如何通过简单的生物学结构解决复杂的计算问题，例如，进行预测。最终的目标不是要建立大脑的真正模型，而是发掘可以解决复杂问题的算法。

神经网络的能力来自它可以从输入数据中学习，对未来进行预测：在这个意义上说，神经网络学习了一种对应关系。数学上说，这种能力是一种有普适性的近似算法。神经网络的预测能力来自网络的分层或多层结构：这种结构可以找出不同尺度或分辨率下的不同特征，将其组合成更高级别的特征。例如，从线条到线条的集合到形状。

6.3 神经元

神经网络由人工神经元组成：这些神经元有计算能力，使用激活函数，利用输入和权重，输出一个标量。



6.3.1 神经元权重

线性回归的权重和这里的权重类似：每个神经元也有一个误差项，永远是1.0，必须被加权。例如，一个神经元有2个输入值，那就需要3个权重项：一个输入一个权重，加上一个误差项的权重。

权重项的初始值一般是小随机数，例如，0~0.3；也有更复杂的初始化方法。和线性回归一样，权重越大代表网络越复杂，越不稳定。我们希望让权重变小，为此可以使用正则化。

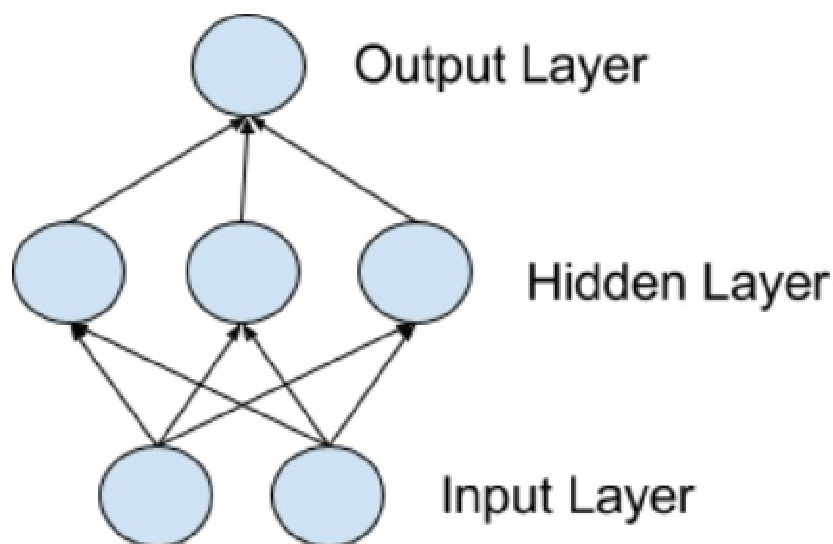
6.3.2 激活函数

神经元的所有输入都被加权求和，输入激活函数中。激活函数就是输入的加权求和值到信号输出的映射。激活函数得名于其功能：控制激活哪个神经元，以及输出信号强度。历史上的激活函数是个阈值：例如，输入加权求和超过0.5，则输出1；反之输出0.0。

激活函数一般使用非线性函数，这样输入的组合方式可以更复杂，提供更多功能。非线性函数可以输出一个分布：例如，逻辑函数（也称为S型函数）输出一个0到1之间的S形分布，正切函数可以输出一个-1到1之间的S形分布。最近的研究表明，线性整流函数的效果更好。

6.4 神经元网络

神经元可以组成网络：每行的神经元叫做一层，一个神经网络可以有很多层。神经网络的结构叫做网络拓扑。



6.4.1 输入层

神经网络最底下的那层叫输入层，因为直接和数据连接。一般的节点数是数据的列数。这层的神经元只将数据传输到下一层。

6.4.2 隐层

输入节点后的层叫隐层，因为不直接和外界相连。最简单的网络中，隐层只有一个神经元，直接输出结果。随着算力增加，现在可以训练很复杂，层数很高的神经网络：历史上需要几辈子才能训练的网络，现在有可能几分钟就能训练好。

6.4.3 输出层

神经网络的最后一层叫输出层，输出问题需要的值。这层的激活函数由问题的类型而定：

- 简单的回归问题：有可能只有一个神经元，没有激活函数
- 两项的分类问题：有可能只有一个神经元，激活函数是S型函数，输出一个0到1之间的概率，代表主类别的概率。也可以用0.5作为阈值：低于0.5输出0，大于0.5输出1.
- 多项的分类问题：有可能有多个神经元，每个代表一类（例如，3个神经元，代表3种不同的鸢尾花 - 这是个经典问题）。激活函数可以使用Softmax函数，每个输出代表是某个类别的概率。最有可能的类别就是输出最高的那组。

6.5 网络训练

6.5.1 预备数据

预处理一下数据：数据必须是数值，例如，实数。如果某一项是类别，需要通过独热编码将其变成数字：对于N种可能的类别，加入N列，对其取0或1代表是否属于该类别。

独热编码也可以对多个类别进行编码：建立一个二进制向量表示类别，输出的结果可以对类别进行分类。神经网络需要所有的数据单位差不多：例如，将所有的数据缩放到0和1之间，这步叫归一化。或者，对数据进行缩放（正则化），让每列的平

均值为0，标准差为1。图像的像素数据也应该这样处理。文字输入可以转化为数字，例如某个单词出现的频率，或者用其他的什么办法转化。

6.5.2 随机梯度下降

随机梯度下降很经典，现在还是很流行。使用的方式是正向传递：每次对网络输入一行数据，激活每层神经元，得出一个输出值。对数据进行预测也用这种方式。

我们把输出和预计值进行比较，算出误差；这个错误通过网络反向传播，更新权重数据。这个算法叫反向传播算法。我们在所有的训练数据上重复此过程，每次网络全部更新叫一轮。神经网络可以训练几十乃至成千上万轮。

6.5.3 权重更新

神经网络的权重可以每次训练都更新，这种方式叫在线更新，速度很快但是有可能造成灾难性结果。或者也可以保存误差数据，最后只更新一次：这种更新叫批量更新，一般而言更稳妥。

因为数据集有可能很大，为了计算速度，每次更新的数据量一般不大，只有几十到几百个数据。权重的更新数量由学习速率（步长）这个参数控制，规定神经网络针对错误的更新速度。这个参数一般很小，0.1或者0.01，乃至更小。也可以调整其他参数：

- 动量：如果上次和这次的方向一样，则加速变化，即使这次的错误不那么大。用于加速网络训练。
- 学习速率衰减：随着训练次数增加而减少学习速率。在一开始加速训练，后面微调参数。

6.5.4 进行预测

训练好的神经网络就可以进行预测了。可以使用测试数据进行测量，看看能不能预测新的数据；也可以部署网络，进行预测。网络只需要保存拓扑结构和最终的权重。将新的数据喂进去，经过前向传输，神经网络就会做出预测了。

6.6 总结

本章关于利用人工神经网络进行机器学习。总结一下：

- 神经网络不是大脑的模型，而是计算模型，可以解决复杂的机器学习问题
- 神经网络由带有权重和激活函数的神经元组成
- 神经网络分层，用随机梯度下降训练
- 应该预处理数据

6.6.1 下一章

你已经了解了神经网络：下一章我们用Keras动手制作第一个神经网络。

第7章 使用Keras开发神经网络

Keras基于Python，开发深度学习模型很容易。Keras将Theano和TensorFlow的数值计算封装好，几句话就可以配置并训练神经网络。本章开始使用Keras开发神经网络。本章将：

- 将CSV数据读入Keras
- 用Keras配置并编译多层感知器模型
- 用验证数据集验证Keras模型

我们开始吧。

7.1 简介

虽然代码量不大，但是我们还是慢慢来。大体分几步：

1. 导入数据
2. 定义模型
3. 编译模型
4. 训练模型
5. 测试模型
6. 写出程序

7.2 皮马人糖尿病数据集

我们使用皮马人糖尿病数据集（Pima Indians onset of diabetes），在UCI的机器学习网站可以免费下载。数据集的内容是皮马人的医疗记录，以及过去5年内是否有糖尿病。所有的数据都是数字，问题是（是否有糖尿病是1或0），是二分类问题。数据的数量级不同，有8个属性：

1. 怀孕次数
2. 2小时口服葡萄糖耐量试验中的血浆葡萄糖浓度
3. 舒张压（毫米汞柱）
4. 2小时血清胰岛素（mu U/ml)
5. 体重指数（BMI）
6. 糖尿病血系功能
7. 年龄（年）
8. 类别：过去5年内是否有糖尿病

所有的数据都是数字，可以直接导入Keras。本书后面也会用到这个数据集。数据有768行，前5行的样本长这样：

```
6,148,72,35,0,33.6,0.627,50,1
1,85,66,29,0,26.6,0.351,31,0
8,183,64,0,0,23.3,0.672,32,1
1,89,66,23,94,28.1,0.167,21,0
0,137,40,35,168,43.1,2.288,33,1
```

数据在本书代码的 `data` 目录，也可以在UCI机器学习的网站下载。把数据和Python文件放在一起，改名：

```
pima-indians-diabetes.csv
```

基准准确率是65.1%，在10次交叉验证中最高的正确率是77.7%。在UCI机器学习的网站可以得到数据集的更多资料。

7.3 导入资料

使用随机梯度下降时最好固定随机数种子，这样你的代码每次运行的结果都一致。这种做法在演示结果、比较算法或debug时特别有效。你可以随便选种子：

```
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
```

现在导入皮马人数据集。NumPy的 `loadtxt()` 函数可以直接带入数据，输入变量是8个，输出1个。导入数据后，我们把数据分成输入和输出两组以便交叉检验：

```
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter="
,")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
```

这样我们的数据每次结果都一致，可以定义模型了。

7.4 定义模型

Keras的模型由层构成：我们建立一个 `Sequential` 模型，一层层加入神经元。第一步是确定输入层的数目正确：在创建模型时用 `input_dim` 参数确定。例如，有8个输入变量，就设成8。

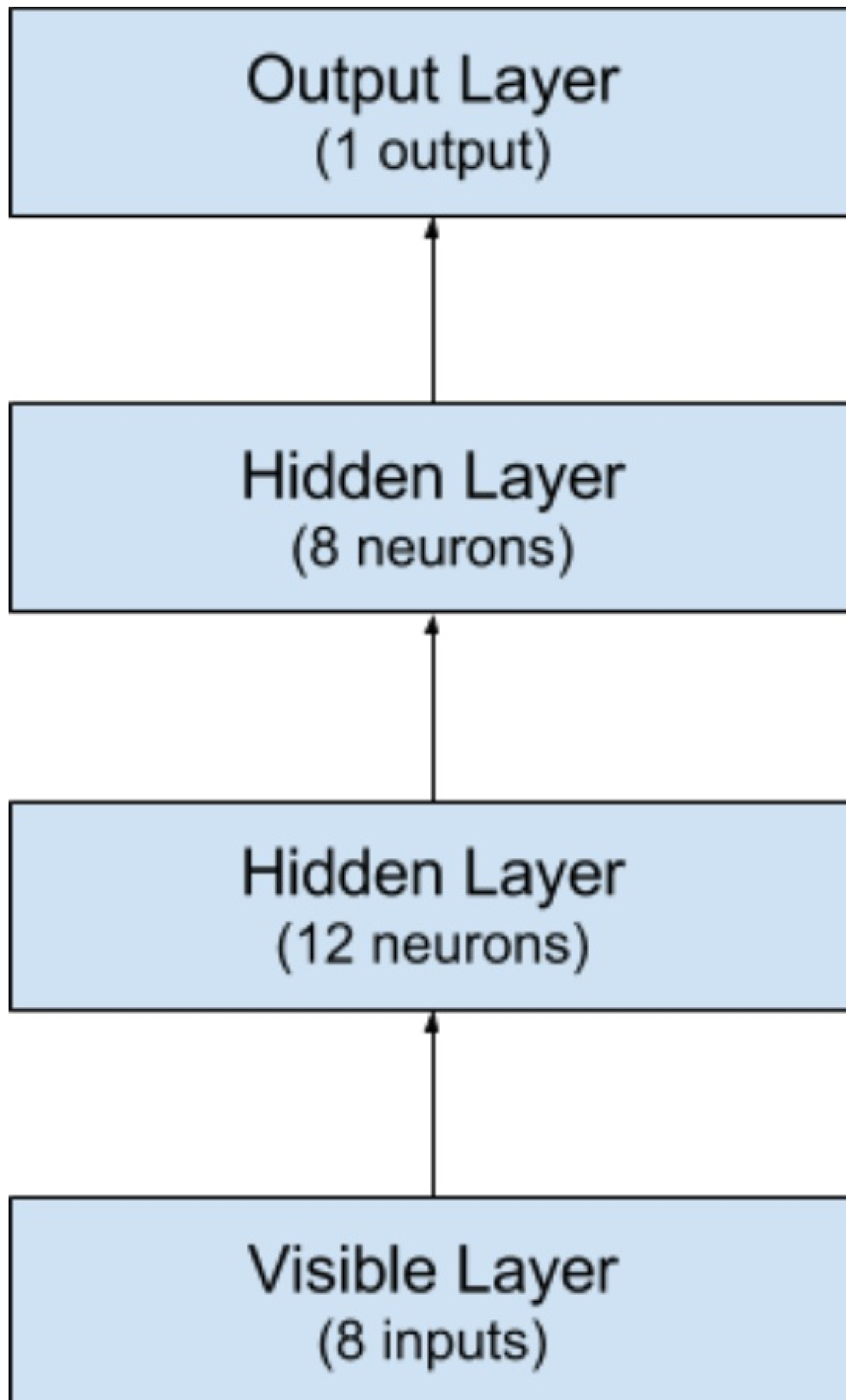
隐层怎么设置？这个问题很难回答，需要慢慢试验。一般来说，如果网络够大，即使存在问题也不会有影响。这个例子里我们用3层全连接网络。

全连接层用 `Dense` 类定义：第一个参数是本层神经元个数，然后是初始化方式和激活函数。这里的初始化方法是0到0.05的连续型均匀分布（ `uniform` ），Keras的默认方法也是这个。也可以用高斯分布进行初始化（ `normal` ）。

前两层的激活函数是线性整流函数（ `relu` ），最后一层的激活函数是S型函数（ `sigmoid` ）。之前大家喜欢用S型和正切函数，但现在线性整流函数效果更好。为了保证输出是0到1的概率数字，最后一层的激活函数是S型函数，这样映射到0.5的阈值函数也容易。前两个隐层分别有12和8个神经元，最后一层是1个神经元（是否有糖尿病）。

```
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, init='uniform', activation='rel
u')) model.add(Dense(8, init='uniform', activation='relu')) mode
l.add(Dense(1, init='uniform', activation='sigmoid'))
```

网络的结构如图：



7.5 编译模型

定义好的模型可以编译：Keras会调用Theano或者TensorFlow编译模型。后端会自动选择表示网络的最佳方法，配合你的硬件。这步需要定义几个新的参数。训练神经网络的意义是：找到最好的一组权重，解决问题。

我们需要定义损失函数和优化算法，以及需要收集的数据。我们使用`binary_crossentropy`，错误的对数作为损失函数；`adam`作为优化算法，因为这东西好用。想深入了解请查阅：Adam: A Method for Stochastic Optimization论文。因为这个问题是分类问题，我们收集每轮的准确率。

7.6 训练模型

终于开始训练了！调用模型的`fit()`方法即可开始训练。

网络按轮训练，通过`nb_epoch`参数控制。每次送入的数据（批尺寸）可以用`batch_size`参数控制。这里我们只跑150轮，每次10个数据。多试试就知道了。

```
# Fit the model
model.fit(X, Y, nb_epoch=150, batch_size=10)
```

现在CPU或GPU开始煎鸡蛋了。

7.7 测试模型

我们把测试数据拿出来检验一下模型的效果。注意这样不能测试在新数据的预测能力。应该将数据分成训练和测试集。

调用模型的`evaluation()`方法，传入训练时的数据。输出是平均值，包括平均误差和其他的数据，例如准确度。

```
# evaluate the model
scores = model.evaluate(X, Y)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

7.8 写出程序

用Keras做机器学习就是这么简单。我们把代码放在一起：

```
# Create first network with Keras
from keras.models import Sequential
from keras.layers import Dense
import numpy
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter="
,")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, init='uniform', activation='rel
u')) model.add(Dense(8, init='uniform', activation='relu'))
model.add(Dense(1, init='uniform', activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metr
ics=['accuracy']) # Fit the model
model.fit(X, Y, nb_epoch=150, batch_size=10)
# evaluate the model
scores = model.evaluate(X, Y)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

训练时每轮会输出一次损失和正确率，以及最终的效果。在我的CPU上用Theano大约跑10秒：

```
...
Epoch 143/150
768/768 [==============================] - 0s - loss: 0.4614 - a
cc: 0.7878
Epoch 144/150
768/768 [==============================] - 0s - loss: 0.4508 - a
cc: 0.7969
Epoch 145/150
768/768 [==============================] - 0s - loss: 0.4580 - a
cc: 0.7747
Epoch 146/150
768/768 [==============================] - 0s - loss: 0.4627 - a
cc: 0.7812
Epoch 147/150
768/768 [==============================] - 0s - loss: 0.4531 - a
cc: 0.7943
Epoch 148/150
768/768 [==============================] - 0s - loss: 0.4656 - a
cc: 0.7734
Epoch 149/150
768/768 [==============================] - 0s - loss: 0.4566 - a
cc: 0.7839
Epoch 150/150
768/768 [==============================] - 0s - loss: 0.4593 - a
cc: 0.7839
768/768 [==============================] - 0s
acc: 79.56%
```

7.9 总结

本章关于利用Keras创建神经网络。总结一下：

- 如何导入数据
- 如何用Keras定义神经网络
- 如何调用后端编译模型
- 如何训练模型
- 如何测试模型

7.9.1 下一章

现在你已经知道如何如何用Keras开发神经网络：下一章讲讲如何在新的数据上进行测试。

第8章 测试神经网络

深度学习有很多参数要调：大部分都是拍脑袋的。所以测试特别重要：本章我们讨论几种测试方法。本章将：

- 使用Keras进行自动验证
- 使用Keras进行手工验证
- 使用Keras进行K折交叉验证

我们开始吧。

8.1 口算神经网络

创建神经网络时有很多参数：很多时候可以从别人的网络上抄，但是最终还是需要一点点做实验。无论是网络的拓扑结构（层数、大小、每层类型）还是小参数（损失函数、激活函数、优化算法、训练次数）等。

一般深度学习的数据集都很大，数据有几十万乃至几亿个。所以测试方法至关重要。

8.2 分割数据

数据量大和网络复杂会造成训练时间很长，所以需要将数据分成训练、测试或验证数据集。Keras提供两种办法：

1. 自动验证
2. 手工验证

8.2.1 自动验证

Keras可以将数据自动分出一部分，每次训练后进行验证。在训练时用 `validation_split` 参数可以指定验证数据的比例，一般是总数据的20%或者33%。下面的代码在第七章上加入了自动验证：

```
# MLP with automatic validation set
from keras.models import Sequential
from keras.layers import Dense
import numpy
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter="
,")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, init='uniform', activation='rel
u')) model.add(Dense(8, init='uniform', activation='relu'))
model.add(Dense(1, init='uniform', activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metr
ics=['accuracy']) # Fit the model
model.fit(X, Y, validation_split=0.33, nb_epoch=150, batch_size=
10)
```

训练时，每轮会显示训练和测试数据的数据：

```
Epoch 145/150
514/514 [==============================] - 0s - loss: 0.4885 - a
cc: 0.7743 - val_loss:
    0.5016 - val_acc: 0.7638
Epoch 146/150
514/514 [==============================] - 0s - loss: 0.4862 - a
cc: 0.7704 - val_loss:
    0.5202 - val_acc: 0.7323
Epoch 147/150
514/514 [==============================] - 0s - loss: 0.4959 - a
cc: 0.7588 - val_loss:
    0.5012 - val_acc: 0.7598
Epoch 148/150
514/514 [==============================] - 0s - loss: 0.4966 - a
cc: 0.7665 - val_loss:
    0.5244 - val_acc: 0.7520
Epoch 149/150
514/514 [==============================] - 0s - loss: 0.4863 - a
cc: 0.7724 - val_loss:
    0.5074 - val_acc: 0.7717
Epoch 150/150
514/514 [==============================] - 0s - loss: 0.4884 - a
cc: 0.7724 - val_loss:
    0.5462 - val_acc: 0.7205
```

8.2.2 手工验证

Keras也可以手工进行验证。我们定义一个 `train_test_split` 函数，将数据分成2：1的测试和验证数据集。在调用 `fit()` 方法时需要加入 `validation_data` 参数作为验证数据，数组的项目分别是输入和输出数据。

```
# MLP with manual validation set
from keras.models import Sequential
from keras.layers import Dense
from sklearn.cross_validation import train_test_split
import numpy
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter="
,")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# split into 67% for train and 33% for test
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_s
ize=0.33, random_state=seed) # create model
model = Sequential()
model.add(Dense(12, input_dim=8, init='uniform', activation='rel
u'))
model.add(Dense(8, init='uniform', activation='relu'))
model.add(Dense(1, init='uniform', activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metr
ics=['accuracy'])
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test,y_test), nb_
epoch=150, batch_size=10)
```

和自动化验证一样，每轮训练后，Keras会输出训练和验证结果：

```
...
Epoch 145/150
514/514 [==============================] - 0s - loss: 0.5001 - a
cc: 0.7685 - val_loss:
    0.5617 - val_acc: 0.7087
Epoch 146/150
514/514 [==============================] - 0s - loss: 0.5041 - a
cc: 0.7529 - val_loss:
    0.5423 - val_acc: 0.7362
Epoch 147/150
514/514 [==============================] - 0s - loss: 0.4936 - a
cc: 0.7685 - val_loss:
    0.5426 - val_acc: 0.7283
Epoch 148/150
514/514 [==============================] - 0s - loss: 0.4957 - a
cc: 0.7685 - val_loss:
    0.5430 - val_acc: 0.7362
Epoch 149/150
514/514 [==============================] - 0s - loss: 0.4953 - a
cc: 0.7685 - val_loss:
    0.5403 - val_acc: 0.7323
Epoch 150/150
514/514 [==============================] - 0s - loss: 0.4941 - a
cc: 0.7743 - val_loss:
    0.5452 - val_acc: 0.7323
```

8.3 手工K折交叉验证

机器学习的金科玉律是K折验证，以验证模型对未来数据的预测能力。K折验证的方法是：将数据分成K组，留下1组验证，其他数据用作训练，直到每种分发的性能一致。

深度学习一般不用交叉验证，因为对算力要求太高。例如，K折的次数一般是5或者10折：每组都需要训练并验证，训练时间成倍上升。然而，如果数据量小，交叉验证的效果更好，误差更小。

scikit-learn有 `StratifiedKFold` 类，我们用它把数据分成10组。抽样方法是分层抽样，尽可能保证每组数据量一致。然后我们在每组上训练模型，使用 `verbose=0` 参数关闭每轮的输出。训练后，Keras会输出模型的性能，并存储模型。最终，Keras输出性能的平均值和标准差，为性能估算提供更准确的估计：

```
# MLP for Pima Indians Dataset with 10-fold cross validation
from keras.models import Sequential
from keras.layers import Dense
from sklearn.cross_validation import StratifiedKFold
import numpy
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter="
,")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# define 10-fold cross validation test harness
kfold = StratifiedKFold(y=Y, n_folds=10, shuffle=True, random_st
ate=seed)
cvscores = []
for i, (train, test) in enumerate(kfold):
    # create model
    model = Sequential()
    model.add(Dense(12, input_dim=8, init='uniform', activation=
'relu')) model.add(Dense(8, init='uniform', activation='relu'))
    model.add(Dense(1, init='uniform', activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy']) # Fit the model
model.fit(X[train], Y[train], nb_epoch=150, batch_size=10, verbo
se=0)
    # evaluate the model
    scores = model.evaluate(X[test], Y[test], verbose=0)
    print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100)
) cvscores.append(scores[1] * 100)
    print("%.2f%% (+/- %.2f%%)" % (numpy.mean(cvscores), numpy.s
td(cvscores)))
```

输出是：

```
acc: 77.92%
acc: 79.22%
acc: 76.62%
acc: 77.92%
acc: 75.32%
acc: 74.03%
acc: 77.92%
acc: 71.43%
acc: 71.05%
acc: 75.00%
75.64% (+/- 2.67%)
```

每次循环都需要重新生成模型，使用对应的数据训练。下一章我们用scikit-learn直接使用Keras的模型。

8.4 总结

本章关于测试神经网络的性能。总结一下：

- 如何自动将数据分成训练和测试组
- 如何人工对数据分组
- 如何使用K折法测试性能

8.4.1 下一章

现在你已经知道如何如何测试神经网络的性能：下一章讲讲如何在scikit-learn中直接使用Keras的模型。

第9章 使用Scikit-Learn调用Keras的模型

scikit-learn是最受欢迎的Python机器学习库。本章我们将使用scikit-learn调用Keras生成的模型。本章将：

- 使用scikit-learn封装Keras的模型
- 使用scikit-learn对Keras的模型进行交叉验证
- 使用scikit-learn，利用网格搜索调整Keras模型的超参

我们开始吧。

9.1 简介

Keras在深度学习很受欢迎，但是只能做深度学习：Keras是最小化的深度学习库，目标在于快速搭建深度学习模型。基于SciPy的scikit-learn，数值运算效率很高，适用于普遍的机器学习任务，提供很多机器学习工具，包括但不限于：

- 使用K折验证模型
- 快速搜索并测试超参

Keras为scikit-learn封装了 `KerasClassifier` 和 `KerasRegressor` 。本章我们继续使用第7章的模型。

9.2 使用交叉验证检验深度学习模型

Keras的 `KerasClassifier` 和 `KerasRegressor` 两个类接受 `build_fn` 参数，传入编译好的模型。我们加入 `nb_epoch=150` 和 `batch_size=10` 这两个参数：这两个参数会传入模型的 `fit()` 方法。我们用scikit-learn的 `StratifiedKFold` 类进行10折交叉验证，测试模型在未知数据的性能，并使用 `cross_val_score()` 函数检测模型，打印结果。

```
# MLP for Pima Indians Dataset with 10-fold cross validation via
 sklearn
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.cross_validation import StratifiedKFold
from sklearn.cross_validation import cross_val_score
import numpy
import pandas
# Function to create model, required for KerasClassifier
def create_model():
    # create model
    model = Sequential()
    model.add(Dense(12, input_dim=8, init='uniform', activation=
'relu')) model.add(Dense(8, init='uniform', activation='relu'))
model.add(Dense(1, init='uniform', activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy']) return model
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter="
,")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = KerasClassifier(build_fn=create_model, nb_epoch=150, bat
ch_size=10)
# evaluate using 10-fold cross validation
kfold = StratifiedKFold(y=Y, n_folds=10, shuffle=True, random_st
ate=seed)
results = cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

每轮训练会输出一次结果，加上最终的平均性能：

```
...
Epoch 145/150
692/692 [==============================] - 0s - loss: 0.4671 - a
cc: 0.7803
Epoch 146/150
692/692 [==============================] - 0s - loss: 0.4661 - a
cc: 0.7847
Epoch 147/150
692/692 [==============================] - 0s - loss: 0.4581 - a
cc: 0.7803
Epoch 148/150
692/692 [==============================] - 0s - loss: 0.4657 - a
cc: 0.7688
Epoch 149/150
692/692 [==============================] - 0s - loss: 0.4660 - a
cc: 0.7659
Epoch 150/150
692/692 [==============================] - 0s - loss: 0.4574 - a
cc: 0.7702
76/76 [==============================] - 0s
0.756442244065
```

比起手工测试，使用scikit-learn容易的多。

9.3 使用网格搜索调整深度学习模型的参数

使用scikit-learn封装Keras的模型十分简单。进一步想：我们可以给 `fit()` 方法传入参数， `KerasClassifier` 的 `build_fn` 方法也可以传入参数。可以利用这点进一步调整模型。

我们用网格搜索测试不同参数的性能： `create_model()` 函数可以传入 `optimizer` 和 `init` 参数，虽然都有默认值。那么我们可以用不同的优化算法和初始权重调整网络。具体说，我们希望搜索：

- 优化算法：搜索权重的方法
- 初始权重：初始化不同的网络
- 训练次数：对模型训练的次数
- 批次大小：每次训练的数据量

所有的参数组成一个字典，传入scikit-learn的 `GridSearchCV` 类： `GridSearchCV` 会对每组参数（2×3×3×3）进行训练，进行3折交叉检验。

计算量巨大：耗时巨长。如果模型小还可以取一部分数据试试。第7章的模型可以用，因为网络和数据集都不大（1000个数据内，9个参数）。最后scikit-learn会输出最好的参数和模型，以及平均值。

```
# MLP for Pima Indians Dataset with grid search via sklearn
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.grid_search import GridSearchCV
import numpy
import pandas
# Function to create model, required for KerasClassifier
def create_model(optimizer='rmsprop', init='glorot_uniform'):
    # create model
    model = Sequential()
    model.add(Dense(12, input_dim=8, init=init, activation='relu'
)) model.add(Dense(8, init=init, activation='relu')) model.add(D
ense(1, init=init, activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer=optimize
r, metrics=['accuracy']) return model
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter="
,")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = KerasClassifier(build_fn=create_model)
# grid search epochs, batch size and optimizer
optimizers = ['rmsprop', 'adam']
init = ['glorot_uniform', 'normal', 'uniform']
epochs = numpy.array([50, 100, 150])
batches = numpy.array([5, 10, 20])
param_grid = dict(optimizer=optimizers, nb_epoch=epochs, batch_s
ize=batches, init=init) grid = GridSearchCV(estimator=model, par
am_grid=param_grid)
grid_result = grid.fit(X, Y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_resul
t.best_params_))
for params, mean_score, scores in grid_result.grid_scores_:
    print("%f (%f) with: %r" % (scores.mean(), scores.std(), par
ams))
```

用CPU差不多要5分钟，结果如下。我们发现使用均匀分布初始化，`rmsprop` 优化算法，150轮，批尺寸为5时效果最好，正确率约75%：

```
Best: 0.751302 using {'init': 'uniform', 'optimizer': 'rmsprop',
'nb_epoch': 150, 'batch_size': 5}
0.653646 (0.031948) with: {'init': 'glorot_uniform', 'optimizer'
: 'rmsprop', 'nb_epoch': 50, 'batch_size': 5}
0.665365 (0.004872) with: {'init': 'glorot_uniform', 'optimizer'
: 'adam', 'nb_epoch': 50, 'batch_size': 5}
0.683594 (0.037603) with: {'init': 'glorot_uniform', 'optimizer'
: 'rmsprop', 'nb_epoch': 100, 'batch_size': 5}
0.709635 (0.034987) with: {'init': 'glorot_uniform', 'optimizer'
: 'adam', 'nb_epoch': 100, 'batch_size': 5}
0.699219 (0.009568) with: {'init': 'glorot_uniform', 'optimizer'
: 'rmsprop', 'nb_epoch': 150, 'batch_size': 5}
0.725260 (0.008027) with: {'init': 'glorot_uniform', 'optimizer'
: 'adam', 'nb_epoch': 150, 'batch_size': 5}
0.686198 (0.024774) with: {'init': 'normal', 'optimizer': 'rmspr
op', 'nb_epoch': 50, 'batch_size': 5}
0.718750 (0.014616) with: {'init': 'normal', 'optimizer': 'adam'
, 'nb_epoch': 50, 'batch_size': 5}
0.725260 (0.028940) with: {'init': 'normal', 'optimizer': 'rmspr
op', 'nb_epoch': 100, 'batch_size': 5}
0.727865 (0.028764) with: {'init': 'normal', 'optimizer': 'adam'
, 'nb_epoch': 100, 'batch_size': 5}
0.748698 (0.035849) with: {'init': 'normal', 'optimizer': 'rmspr
op', 'nb_epoch': 150, 'batch_size': 5}
0.712240 (0.039623) with: {'init': 'normal', 'optimizer': 'adam'
, 'nb_epoch': 150, 'batch_size': 5}
0.699219 (0.024910) with: {'init': 'uniform', 'optimizer': 'rmsp
rop', 'nb_epoch': 50, 'batch_size': 5}
0.703125 (0.011500) with: {'init': 'uniform', 'optimizer': 'adam'
, 'nb_epoch': 50, 'batch_size': 5}
0.720052 (0.015073) with: {'init': 'uniform', 'optimizer': 'rmsp
rop', 'nb_epoch': 100, 'batch_size': 5}
0.712240 (0.034987) with: {'init': 'uniform', 'optimizer': 'adam'
, 'nb_epoch': 100, 'batch_size': 5}
0.751302 (0.031466) with: {'init': 'uniform', 'optimizer': 'rmsp
rop', 'nb_epoch': 150, 'batch_size': 5}
0.734375 (0.038273) with: {'init': 'uniform', 'optimizer': 'adam'
, 'nb_epoch': 150, 'batch_size': 5}
...
```

9.4 总结

本章关于使用scikit-learn封装并测试神经网络的性能。总结一下：

- 如何使用scikit-learn封装Keras模型
- 如何使用scikit-learn测试Keras模型的性能
- 如何使用scikit-learn调整Keras模型的超参

使用scikit-learn调整参数比手工调用Keras简便的多。

9.4.1 下一章

现在你已经知道如何如何在scikit-learn调用Keras模型：可以开工了。接下来几章我们会用Keras创造不同的端到端模型，从多类分类问题开始。

第10章 项目：多类花朵分类

本章我们使用Keras为多类分类开发并验证一个神经网络。本章包括：

- 将CSV导入Keras
- 为Keras预处理数据
- 使用scikit-learn验证Keras模型

我们开始吧。

10.1 鸢尾花分类数据集

本章我们使用经典的鸢尾花数据集。这个数据集已经被充分研究过，4个输入变量都是数字，量纲都是厘米。每个数据代表花朵的不同参数，输出是分类结果。数据的属性是（厘米）：

1. 萼片长度
2. 萼片宽度
3. 花瓣长度
4. 花瓣宽度
5. 类别

这个问题是多类分类的：有两种以上的类别需要预测，确切的说，3种。这种问题需要对神经网络做出特殊调整。数据有150条：前5行是：

```
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
5.0,3.6,1.4,0.2,Iris-setosa
```

鸢尾花数据集已经被充分研究，模型的准确率可以达到95%到97%，作为目标很不错。本书的data目录下附带了示例代码和数据，也可以从UCI机器学习网站下载，重命名为 `iris.csv` 。数据集的详情请在UCI机器学习网站查询。

10.2 导入库和函数

我们导入所需要的库和函数，包括深度学习包Keras、数据处理包pandas和模型测试包scikit-learn。

```
import numpy
import pandas
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from keras.utils import np_utils
from sklearn.cross_validation import cross_val_score
from sklearn.cross_validation import KFold
from sklearn.preprocessing import LabelEncoder
from sklearn.pipeline import Pipeline
```

10.3 指定随机数种子

我们指定一个随机数种子，这样重复运行的结果会一致，以便复现随机梯度下降的结果：

```
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
```

10.4 导入数据

数据可以直接导入。因为数据包含字符，用pandas更容易。然后可以将数据的属性（列）分成输入变量（X）和输出变量（Y）：

```
# load dataset
dataframe = pandas.read_csv("iris.csv", header=None)
dataset = dataframe.values
X = dataset[:,0:4].astype(float)
Y = dataset[:,4]
```

10.5 输出变量编码

数据的类型是字符串：在使用神经网络时应该将类别编码成矩阵，每行每列代表所属类别。可以使用独热编码，或者加入一列。这个数据中有3个类别：`Iris-setosa`、`Iris-versicolor`和`Iris-virginica`。如果数据是

```
Iris-setosa
Iris-versicolor
Iris-virginica
```

用独热编码可以编码成这种矩阵：

```
Iris-setosa, Iris-versicolor, Iris-virginica 1, 0, 0
0, 1, 0
0, 0, 1
```

scikit-learn的 `LabelEncoder` 可以将类别变成数字，然后用Keras的 `to_categorical()` 函数编码：

```
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
encoded_Y = encoder.transform(Y)
# convert integers to dummy variables (i.e. one hot encoded)
dummy_y = np_utils.to_categorical(encoded_Y)
```

10.6 设计神经网络

Keras提供了 `KerasClassifier` ，可以将网络封装，在scikit-learn上用。 `KerasClassifier` 的初始化变量是模型名称，返回供训练的神经网络模型。

我们写一个函数，为鸢尾花分类问题创建一个神经网络：这个全连接网络只有1个带有4个神经元的隐层，和输入的变量数相同。为了效果，隐层使用整流函数作为激活函数。因为我们用了独热编码，网络的输出必须是3个变量，每个变量代表一种花，最大的变量代表预测种类。网络的结构是：

```
4个神经元 输入层 -> [4个神经元 隐层] -> 3个神经元 输出层
```

输出层的函数是S型函数，把可能性映射到概率的0到1。优化算法选择ADAM随机梯度下降，损失函数是对数函数，在Keras中叫 `categorical_crossentropy` ：

```
# define baseline model
def baseline_model():
    # create model
    model = Sequential()
    model.add(Dense(4, input_dim=4, init='normal', activation='r
elu')) model.add(Dense(3, init='normal', activation='sigmoid'))
    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer='ad
am', metrics=['accuracy']) return model
```

可以用这个模型创建 `KerasClassifier` ，也可以传入其他参数，这些参数会传递到 `fit()` 函数中。我们将训练次数 `nb_epoch` 设成150，批尺寸 `batch_size` 设成5， `verbose` 设成0以关闭调试信息：

```
estimator = KerasClassifier(build_fn=baseline_model, nb_epoch=200
, batch_size=5, verbose=0)
```

10.7 用K折交叉检验测试模型

现在可以测试模型效果了。scikit-learn有很多种办法可以测试模型，其中最重要的就是K折检验。我们先设定模型的测试方法：K设为10（默认值很好），在分割前随机重排数据：

```
kfold = KFold(n=len(X), n_folds=10, shuffle=True, random_state=s
eed)
```

这样我们就可以在数据集（ `X` 和 `dummy_y` ）上用10折交叉检验（ `kfold` ）测试性能了。模型需要10秒钟就可以跑完，每次检验输出结果：

```
results = cross_val_score(estimator, X, dummy_y, cv=kfold)
print("Accuracy: %.2f%% (%.2f%%)" % (results.mean()*100, results
.std()*100))
```

输出结果的均值和标准差，这样可以验证模型的预测能力，效果拔群：

```
Baseline: 95.33% (4.27%)
```

10.8 总结

本章关于使用Keras开发深度学习项目。总结一下：

- 如何导入数据
- 如何使用独热编码处理多类分类数据
- 如何与scikit-learn一同使用Keras
- 如何用Keras定义多类分类神经网络
- 如何用scikit-learn通过K折交叉检验测试Keras的模型

10.8.1 下一章

本章完整描述了Keras项目的开发：下一章我们开发一个二分类网络，并调优。

第11章 项目：声呐返回值分类

本章我们使用Keras开发一个二分类网络。本章包括：

- 将数据导入Keras
- 为表格数据定义并训练模型
- 在未知数据上测试Keras模型的性能
- 处理数据以提高准确率
- 调整Keras模型的拓扑和配置

我们开始吧。

11.1 声呐物体分类数据

本章使用声呐数据，包括声呐在不同物体的返回。数据有60个变量，代表不同角度的返回值。目标是将石头和金属筒（矿石）分开。

所有的数据都是连续的，从0到1；输出变量中M代表矿石，R代表石头，需要转换为1和0。数据集有208条数据，在本书的data目录下，也可以自行下载，重命名为 `sonar.csv` 。

此数据集可以作为性能测试标准：我们知道什么程度的准确率代表模型是优秀的。交叉检验后，一般的网络可以达到84%的准确率，最高可以达到88%。关于数据集详情，请到UCI机器学习网站查看。

11.2 简单的神经网络

先创建一个简单的神经网络试试看。导入所有的库和函数：

```
import numpy
import pandas
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.cross_validation import cross_val_score
from sklearn.preprocessing import LabelEncoder
from sklearn.cross_validation import StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
```

初始化随机数种子，这样每次的结果都一样，帮助debug：

```
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
```

用pandas读入数据：前60列是输入变量（X），最后一列是输出变量（Y）。pandas处理带字符的数据比NumPy更容易。

```
# load dataset
dataframe = pandas.read_csv("sonar.csv", header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:60].astype(float)
Y = dataset[:,60]
```

输出变量现在是字符串：需要编码成数字0和1。scikit-learn的 `LabelEncoder` 可以做到：先将数据用 `fit()` 方法导入，然后用 `transform()` 函数编码，加入一列：

```
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
encoded_Y = encoder.transform(Y)
```

现在可以用Keras创建神经网络模型了。我们用scikit-learn进行随机K折验证，测试模型效果。Keras的模型用 `KerasClassifier` 封装后可以在scikit-learn中调用，取的变量建立的模型；其他变量会传入 `fit()` 方法中，例如训练次数和批尺寸。我们写一个函数创建这个模型：只有一个全连接层，神经元数量和输入变量数一样，作为最基础的模型。

模型的权重是比较小的高斯随机数，激活函数是整流函数，输出层只有一个神经元，激活函数是S型函数，代表某个类的概率。损失函数还是对数损失函数（`binary_crossentropy`），这个函数适用于二分类问题。优化算法是Adam随机梯度下降，每轮收集模型的准确率。

```
# baseline model
def create_baseline():
# create model
    model = Sequential()
    model.add(Dense(60, input_dim=60, init='normal', activation=
'relu')) model.add(Dense(1, init='normal', activation='sigmoid')
)
# Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy']) return model
```

用scikit-learn测试一下模型。向 `KerasClassifier` 传入训练次数（默认值），关闭日志：

```
# evaluate model with standardized dataset
estimator = KerasClassifier(build_fn=create_baseline, nb_epoch=1
00, batch_size=5, verbose=0)
kfold = StratifiedKFold(y=encoded_Y, n_folds=10, shuffle=True, r
andom_state=seed)
results = cross_val_score(estimator, X, encoded_Y, cv=kfold)
print("Results: %.2f%% (%.2f%%)" % (results.mean()*100, results.
std()*100))
```

输出内容是测试的平均数和标准差。

```
Baseline: 81.68% (5.67%)
```

不用很累很麻烦效果也可以很好。

11.3 预处理数据以增加性能

预处理数据是个好习惯。神经网络喜欢输入类型的比例和分布一致，为了达到这点可以使用正则化，让数据的平均值是0，标准差是1，这样可以保留数据的分布情况。

scikit-learn的 `StandardScaler` 可以做到这点。不应该在整个数据集上直接应用正则化：应该只在测试数据上交叉验证时进行正则化处理，使正则化成为交叉验证的一环，让模型没有新数据的先验知识，防止模型发散。

scikit-learn的 `Pipeline` 可以直接做到这些。我们先定义一个 `StandardScaler` ，然后进行验证：

```
# evaluate baseline model with standardized dataset
numpy.random.seed(seed)
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(build_fn=create_baseli
ne, nb_epoch=100,
    batch_size=5, verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(y=encoded_Y, n_folds=10, shuffle=True, r
andom_state=seed)
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)
print("Standardized: %.2f%% (%.2f%%)" % (results.mean()*100, res
ults.std()*100))
```

结果如下，平均效果有一点进步。

```
Standardized: 84.07% (6.23%)
```

11.4 调整模型的拓扑和神经元

神经网络有很多参数，例如初始化权重、激活函数、优化算法等等。我们一直没有说到调整网络的拓扑结构：扩大或缩小网络。我们试验一下：

11.4.1 缩小网络

有可能数据中有冗余：原始数据是不同角度的信号，有可能其中某些角度有相关性。我们把第一层隐层缩小一些，强行提取特征试试。

我们把之前的模型隐层的60个神经元减半到30个，这样神经网络需要挑选最重要的信息。之前的正则化有效果：我们也一并做一下.

```
# smaller model
def create_smaller():
    # create model
    model = Sequential()
    model.add(Dense(30, input_dim=60, init='normal', activation=
'relu')) model.add(Dense(1, init='normal', activation='sigmoid')
)
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy']) return model
numpy.random.seed(seed)
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(build_fn=create_smalle
r, nb_epoch=100,
    batch_size=5, verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(y=encoded_Y, n_folds=10, shuffle=True, r
andom_state=seed)
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)
print("Smaller: %.2f%% (%.2f%%)" % (results.mean()*100, results.
std()*100))
```

结果如下。平均值有少许提升，方差减少很多：这么做果然有效，因为这次的训练时间只需要之前的一半！

```
Smaller: 84.61% (4.65%)
```

11.4.2 扩大网络

扩大网络后，神经网络更有可能提取关键特征，以非线性方式组合。我们对之前的网络简单修改一下：在原来的隐层后加入一层30个神经元的隐层。现在的网络是：

```
60 inputs -> [60 -> 30] -> 1 output
```

我们希望在缩减信息前可以对所有的变量建模，和缩小网络时的想法类似。这次我们加一层，帮助网络挑选信息：

```
# larger model
def create_larger():
    # create model
    model = Sequential()
    model.add(Dense(60, input_dim=60, init='normal', activation=
'relu')) model.add(Dense(30, init='normal', activation='relu'))
model.add(Dense(1, init='normal', activation='sigmoid'))
     # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy']) return model
numpy.random.seed(seed)
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(build_fn=create_larger
, nb_epoch=100,
    batch_size=5, verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(y=encoded_Y, n_folds=10, shuffle=True, r
andom_state=seed)
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)
print("Larger: %.2f%% (%.2f%%)" % (results.mean()*100, results.s
td()*100))
```

这次的结果好了很多，几乎达到业界最优。

```
Larger: 86.47% (3.82%)
```

继续微调网络的结果会更好。你能做到如何？

11.5 总结

本章关于使用Keras开发二分类深度学习项目。总结一下：

- 如何导入数据
- 如何创建基准模型
- 如何用scikit-learn通过K折随机交叉检验测试Keras的模型
- 如何预处理数据
- 如何微调网络

11.5.1 下一章

多分类和二分类介绍完了：下一章是回归问题。

第12章 项目：波士顿住房价格回归

本章关于如何使用Keras和社交网络解决回归问题。本章将：

- 导入CSV数据
- 创建回归问题的神经网络模型
- 使用scikit-learn对Keras的模型进行交叉验证
- 预处理数据以增加效果
- 微调网络参数

我们开始吧。

12.1 波士顿住房价格数据

本章我们研究波士顿住房价格数据集，即波士顿地区的住房信息。我们关心的是住房价格，单位是千美金：所以，这个问题是回归问题。数据有13个输入变量，代表房屋不同的属性：

1. CRIM：人均犯罪率
2. ZN：25,000平方英尺以上民用土地的比例
3. INDUS：城镇非零售业商用土地比例
4. CHAS：是否邻近查尔斯河，1是邻近，0是不邻近
5. NOX：一氧化氮浓度（千万分之一）
6. RM：住宅的平均房间数
7. AGE：自住且建于1940年前的房屋比例
8. DIS：到5个波士顿就业中心的加权距离
9. RAD：到高速公路的便捷度指数
10. TAX：每万元的房产税率
11. PTRATIO：城镇学生教师比例
12. B： 1000(Bk − 0.63)2 其中Bk是城镇中黑人比例
13. LSTAT：低收入人群比例
14. MEDV：自住房中位数价格，单位是千元

这个问题已经被深入研究过，所有的数据都是数字。数据的前5行是：

```
0.00632 18.00 2.310 0 0.5380 6.5750 65.20 4.0900 1 296.0 15.30 3
96.90 4.98 24.00
0.02731 0.00 7.070 0 0.4690 6.4210 78.90 4.9671 2 242.0 17.80 39
6.90 9.14 21.60
0.02729 0.00 7.070 0 0.4690 7.1850 61.10 4.9671 2 242.0 17.80 39
2.83 4.03 34.70
0.03237 0.00 2.180 0 0.4580 6.9980 45.80 6.0622 3 222.0 18.70 39
4.63 2.94 33.40
0.06905 0.00 2.180 0 0.4580 7.1470 54.20 6.0622 3 222.0 18.70 39
6.90 5.33 36.20
```

数据在本书的data目录下，也可以自行下载，重命名为 `housing.csv` 。普通模型的均方误差（MSE）大约是20，和方差（SSE）是$4,500美金。关于数据集详情，请到UCI机器学习网站查看。

12.2 简单的神经网络

先创建一个简单的回归神经网络。导入所有的库和函数：

```
import numpy
import pandas
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasRegressor
from sklearn.cross_validation import cross_val_score
from sklearn.cross_validation import KFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
```

源文件是CSV格式，分隔符是空格：可以用pandas导入，然后分成输入（X）和输出（Y）变量。

```
# load dataset
dataframe = pandas.read_csv("housing.csv", delim_whitespace=True
, header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:13]
Y = dataset[:,13]
```

Keras可以把模型封装好，交给scikit-learn使用，方便测试模型。我们写一个函数，创建神经网络。

代码如下。有一个全连接层，神经元数量和输入变量数一致（13），激活函数还是整流函数。输出层没有激活函数，因为在回归问题中我们希望直接取结果。

优化函数是Adam，损失函数是MSE，和我们要优化的函数一致：这样可以对模型的预测有直观的理解，因为MSE乘方就是千美元计的误差。

```
# define base mode
def baseline_model():
    # create model
    model = Sequential()
    model.add(Dense(13, input_dim=13, init='normal', activation=
'relu')) model.add(Dense(1, init='normal'))
      # Compile model
    model.compile(loss='mean_squared_error', optimizer='adam') r
eturn model
```

使用 `KerasRegressor` 封装这个模型，任何其他的变量都会传入 `fit()` 函数中，例如训练次数和批次大小，这里我们取默认值。老规矩，为了可以复现结果，指定一下随机数种子：

```
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# evaluate model with standardized dataset
estimator = KerasRegressor(build_fn=baseline_model, nb_epoch=100
, batch_size=5, verbose=0)
```

可以测试一下基准模型的结果了：用10折交叉检验看看。

```
kfold = KFold(n=len(X), n_folds=10, random_state=seed)
results = cross_val_score(estimator, X, Y, cv=kfold)
print("Results: %.2f (%.2f) MSE" % (results.mean(), results.std(
)))
```

结果是10次检验的误差均值和标准差。

```
Results: 38.04 (28.15) MSE
```

12.3 预处理数据以增加性能

这个数据集的特点是变量的尺度不一致，所以标准化很有用。

scikit-learn的 `Pipeline` 可以直接进行均一化处理并交叉检验，这样模型不会预先知道新的数据。代码如下：

```
# evaluate model with standardized dataset
numpy.random.seed(seed)
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasRegressor(build_fn=baseline_model
, nb_epoch=50,
    batch_size=5, verbose=0)))
pipeline = Pipeline(estimators)
kfold = KFold(n=len(X), n_folds=10, random_state=seed)
results = cross_val_score(pipeline, X, Y, cv=kfold)
print("Standardized: %.2f (%.2f) MSE" % (results.mean(), results
.std()))
```

效果直接好了一万刀：

```
Standardized: 28.24 (26.25) MSE
```

也可以将数据标准化，在最后一层用S型函数作为激活函数，将比例拉到一样。

12.4 调整模型的拓扑

神经网络有很多可调的参数：最可玩的是网络的结构。这次我们用一个更深的和一个更宽的模型试试。

12.4.1 更深的模型

增加神经网络的层数可以提高效果，这样模型可以提取并组合更多的特征。我们试着加几层隐层：加几句话就行。代码从上面复制下来，在第一层后加一层隐层，神经元数量是上层的一半：

```
def larger_model():
    # create model
    model = Sequential()
    model.add(Dense(13, input_dim=13, init='normal', activation=
'relu')) model.add(Dense(6, init='normal', activation='relu')) m
odel.add(Dense(1, init='normal'))
      # Compile model
    model.compile(loss='mean_squared_error', optimizer='adam') r
eturn model
```

这样的结构是：

```
13 inputs -> [13 -> 6] -> 1 output
```

测试的方法一样，数据正则化一下：

```
numpy.random.seed(seed)
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasRegressor(build_fn=larger_model,
nb_epoch=50, batch_size=5,
    verbose=0)))
pipeline = Pipeline(estimators)
kfold = KFold(n=len(X), n_folds=10, random_state=seed)
results = cross_val_score(pipeline, X, Y, cv=kfold)
print("Larger: %.2f (%.2f) MSE" % (results.mean(), results.std()
))
```

效果好了一点，MSE从28变成24：

```
Larger: 24.60 (25.65) MSE
```

12.4.1 更宽的模型

加宽模型可以增加网络容量。我们减去一层，把隐层的神经元数量加大，从13加到20：

```
def wider_model():
    # create model
    model = Sequential()
    model.add(Dense(20, input_dim=13, init='normal', activation=
'relu')) model.add(Dense(1, init='normal'))
    # Compile model
    model.compile(loss='mean_squared_error', optimizer='adam') r
eturn model
```

网络的结构是：

```
13 inputs -> [20] -> 1 output
```

跑一下试试：

```
numpy.random.seed(seed)
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasRegressor(build_fn=wider_model, n
b_epoch=100, batch_size=5,
    verbose=0)))
pipeline = Pipeline(estimators)
kfold = KFold(n=len(X), n_folds=10, random_state=seed)
results = cross_val_score(pipeline, X, Y, cv=kfold)
print("Wider: %.2f (%.2f) MSE" % (results.mean(), results.std())
)
```

MSE下降到21，效果不错了。

```
Wider: 21.64 (23.75) MSE
```

很难想到，加宽模型比加深模型效果更好：这就是欧皇的力量。

12.5 总结

本章关于使用Keras开发回归深度学习项目。总结一下：

- 如何导入数据
- 如何预处理数据提高性能
- 如何调整网络结构提高性能

12.5.1 下一章

第三部分到此结束：你可以处理一般的机器学习问题了。下一章我们用一些奇技淫巧，使用一些Keras的高级API。

第四部分 Keras与高级多层感知器

第13章 用序列化保存模型

深度学习的模型有可能需要好几天才能训练好，如果没有SL大法就完蛋了。本章关于如何保存和加载模型。本章将：

- 使用HDF5格式保存模型
- 使用JSON格式保存模型
- 使用YAML格式保存模型

我们开始吧。

13.1 简介

Keras中，模型的结构和权重数据是分开的：权重的文件格式是HDF5，这种格式保存数字矩阵效率很高。模型的结构用JSON或YAML导入导出。

本章包括如何手工修改HDF5文件，使用的模型是第7章的皮马人糖尿病模型。

13.1.1 HDF5文件

分层数据格式，版本5（HDF5）可以高效保存大实数矩阵，例如神经网络的权重。HDF5的包需要安装：

```
sudo pip install h5py
```

13.2 使用JSON保存网络结构

JSON的格式很简单，Keras可以用 `to_json()` 把模型导出为JSON格式，再用 `model_from_json()` 加载回来。

模型和权重加载后需要编译一次，让Keras正确调用后端。模型的验证方法和之前一致：

导出：

```python
# MLP for Pima Indians Dataset serialize to JSON and HDF5
from keras.models import Sequential
from keras.layers import Dense
from keras.models import model_from_json
import numpy
import os
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter="
,")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, init='uniform', activation='rel
u')) model.add(Dense(8, init='uniform', activation='relu'))
model.add(Dense(1, init='uniform', activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metr
ics=['accuracy']) # Fit the model
model.fit(X, Y, nb_epoch=150, batch_size=10, verbose=0)
# evaluate the model
scores = model.evaluate(X, Y, verbose=0)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
# serialize model to JSON
model_json = model.to_json()
with open("model.json", "w") as json_file:
    json_file.write(model_json)
# serialize weights to HDF5
model.save_weights("model.h5")
print("Saved model to disk")
```

导入：

```
# later...
# load json and create model
# MLP for Pima Indians Dataset serialize to JSON and HDF5
from keras.models import Sequential
from keras.layers import Dense
from keras.models import model_from_json
import numpy
import os
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter="
,")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, init='uniform', activation='rel
u')) model.add(Dense(8, init='uniform', activation='relu'))
model.add(Dense(1, init='uniform', activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metr
ics=['accuracy']) # Fit the model
model.fit(X, Y, nb_epoch=150, batch_size=10, verbose=0)
# evaluate the model
scores = model.evaluate(X, Y, verbose=0)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
# serialize model to JSON
model_json = model.to_json()
with open("model.json", "w") as json_file:
    json_file.write(model_json)
# serialize weights to HDF5
model.save_weights("model.h5")
print("Saved model to disk")
# later...
# load json and create model
```

结果如下。导入的模型和之前导出时一致：

```
acc: 79.56%
Saved model to disk
Loaded model from disk
acc: 79.56%
```

JSON文件类似：

```
{
    "class_name": "Sequential",
    "config": [{
            "class_name": "Dense",
```

```
                "config": {
                    "W_constraint": null,
                    "b_constraint": null,
                    "name": "dense_1",
                    "output_dim": 12,
                    "activity_regularizer": null,
                    "trainable": true,
                    "init": "uniform",
                    "input_dtype": "float32",
                    "input_dim": 8,
                    "b_regularizer": null,
                    "W_regularizer": null,
                    "activation": "relu",
                    "batch_input_shape": [
                        null,
                        8
                    ]
                }
            },
            {
                "class_name": "Dense",
                "config": {
                    "W_constraint": null,
                    "b_constraint": null,
                    "name": "dense_2",
                    "activity_regularizer": null,
                    "trainable": true,
                    "init": "uniform",
                    "input_dim": null,
                    "b_regularizer": null,
                    "W_regularizer": null,
                    "activation": "relu",
                    "output_dim": 8
                }
            },
            {
                "class_name": "Dense",
                "config": {
                    "W_constraint": null,
                    "b_constraint": null,
                    "name": "dense_3",
                    "activity_regularizer": null,
                    "trainable": true,
                    "init": "uniform",
                    "input_dim": null,
                    "b_regularizer": null,
                    "W_regularizer": null,
                    "activation": "sigmoid",
                    "output_dim": 1
                }
            }
        ]
}
```

13.3 使用YAML保存网络结构

和之前JSON类似，只不过文件格式变成YAML，使用的函数变成了 `to_yaml()` 和 `model_from_yaml()` ：

```
# MLP for Pima Indians Dataset serialize to YAML and HDF5
from keras.models import Sequential
from keras.layers import Dense
from keras.models import model_from_yaml
import numpy
import os
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter="
,")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, init='uniform', activation='rel
u')) model.add(Dense(8, init='uniform', activation='relu'))
model.add(Dense(1, init='uniform', activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metr
ics=['accuracy']) # Fit the model
model.fit(X, Y, nb_epoch=150, batch_size=10, verbose=0)
# evaluate the model
scores = model.evaluate(X, Y, verbose=0)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
# serialize model to YAML
model_yaml = model.to_yaml()
with open("model.yaml", "w") as yaml_file:
    yaml_file.write(model_yaml)
# serialize weights to HDF5
model.save_weights("model.h5")
print("Saved model to disk")


# later...
# load YAML and create model
yaml_file = open('model.yaml', 'r') loaded_model_yaml = yaml_fil
e.read() yaml_file.close()
loaded_model = model_from_yaml(loaded_model_yaml) # load weights
 into new model loaded_model.load_weights("model.h5") print("Loa
ded model from disk")
# evaluate loaded model on test data
loaded_model.compile(loss='binary_crossentropy', optimizer='rmsp
rop', metrics=['accuracy']) score = loaded_model.evaluate(X, Y,
verbose=0)
print "%s: %.2f%%" % (loaded_model.metrics_names[1], score[1]*100
)
```

结果和之前的一样：

```
acc: 79.56%
Saved model to disk
Loaded model from disk
acc: 79.56%
```

YAML文件长这样：

```
class_name: Sequential
config:
- class_name: Dense
  config:
    W_constraint: null
    W_regularizer: null
   activation: relu
    activity_regularizer: null
    b_constraint: null
    b_regularizer: null
    batch_input_shape: !!python/tuple [null, 8]
    init: uniform
    input_dim: 8
    input_dtype: float32
    name: dense_1
    output_dim: 12
    trainable: true
- class_name: Dense
  config: {W_constraint: null, W_regularizer: null, activation:
relu, activity_regularizer:
      null,
    b_constraint: null, b_regularizer: null, init: uniform, inpu
t_dim: null, name: dense_2,
    output_dim: 8, trainable: true}
- class_name: Dense
  config: {W_constraint: null, W_regularizer: null, activation:
sigmoid,
      activity_regularizer: null,
    b_constraint: null, b_regularizer: null, init: uniform, inpu
t_dim: null, name: dense_3,
    output_dim: 1, trainable: true}
```

13.4 总结

本章关于导入导出Keras模型。总结一下：

- 如何用HDF5保存加载权重
- 如何用JSON保存加载模型
- 如何用YAML保存加载模型

13.4.1 下一章

模型可以保存了：下一章关于使用保存点。

第14章 使用保存点保存最好的模型

深度学习有可能需要跑很长时间，如果中间断了（特别是在竞价式实例上跑的时候）就要亲命了。本章关于在训练时中途保存模型。本章将：

- 保存点很重要！
- 每轮打保存点！
- 挑最好的模型！

我们开始吧。

14.1 使用保存点

长时间运行的程序需要能中途保存，加强健壮性。保存的程序应该可以继续运行，或者直接运行。深度学习的保存点用来存储模型的权重：这样可以继续训练，或者直接开始预测。

Keras有回调API，配合 `ModelCheckpoint` 可以每轮保存网络信息，可以定义文件位置、文件名和保存时机等。例如，损失函数或准确率达到某个标准就保存，文件名的格式可以加入时间和准确率等。 `ModelCheckpoint` 需要传入 `fit()` 函数，也需要安装 `h5py` 库。

14.2 效果变好就保存

好习惯：每轮如果效果变好就保存一下。还是用第7章的模型，用33%的数据测试。

每轮后在测试数据集上验证，如果比之前效果好就保存权重（monitor='val_acc', mode='max'）。文件名格式是 `weights-improvement-val_acc=.2f.hdf5` 。

```
# Checkpoint the weights when validation accuracy improves
from keras.models import Sequential
from keras.layers import Dense
from keras.callbacks import ModelCheckpoint
import matplotlib.pyplot as plt
import numpy
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter="
,")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, init='uniform', activation='rel
u'))
model.add(Dense(8, init='uniform', activation='relu'))
model.add(Dense(1, init='uniform', activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metr
ics=['accuracy'])
# checkpoint
filepath="weights-improvement-{epoch:02d}-{val_acc:.2f}.hdf5"
checkpoint = ModelCheckpoint(filepath, monitor='val_acc', verbos
e=1, save_best_only=True,
mode='max')
callbacks_list = [checkpoint]
# Fit the model
model.fit(X, Y, validation_split=0.33, nb_epoch=150, batch_size=
10,
    callbacks=callbacks_list, verbose=0)
```

输出的结果如下：如果效果更好就保存。

```
...
Epoch 00134: val_acc did not improve
Epoch 00135: val_acc did not improve
Epoch 00136: val_acc did not improve
Epoch 00137: val_acc did not improve
Epoch 00138: val_acc did not improve
Epoch 00139: val_acc did not improve
Epoch 00140: val_acc improved from 0.83465 to 0.83858, saving mo
del to
    weights-improvement-140-0.84.hdf5
Epoch 00141: val_acc did not improve
Epoch 00142: val_acc did not improve
Epoch 00143: val_acc did not improve
Epoch 00144: val_acc did not improve
Epoch 00145: val_acc did not improve
Epoch 00146: val_acc improved from 0.83858 to 0.84252, saving mo
del to
    weights-improvement-146-0.84.hdf5
Epoch 00147: val_acc did not improve
Epoch 00148: val_acc improved from 0.84252 to 0.84252, saving mo
del to
    weights-improvement-148-0.84.hdf5
Epoch 00149: val_acc did not improve
```

目录下会保存每次的模型：

```
...
weights-improvement-74-0.81.hdf5
weights-improvement-81-0.82.hdf5
weights-improvement-91-0.82.hdf5
weights-improvement-93-0.83.hdf5
```

这种方法有效，但是文件较多。当然最好的模型肯定保存下来了。

14.3 保存最好的模型

也可以只保存最好的模型：每次如果效果变好就覆盖之前的权重文件，把之前的文件名改成固定的就可以：

```
# Checkpoint the weights for best model on validation accuracy
from keras.models import Sequential
from keras.layers import Dense
from keras.callbacks import ModelCheckpoint
import matplotlib.pyplot as plt
import numpy
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter="
,")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, init='uniform', activation='rel
u')) model.add(Dense(8, init='uniform', activation='relu')) mode
l.add(Dense(1, init='uniform', activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metr
ics=['accuracy'])
# checkpoint
filepath="weights.best.hdf5"
checkpoint = ModelCheckpoint(filepath, monitor='val_acc', verbos
e=1, save_best_only=True,
mode='max')
callbacks_list = [checkpoint]
# Fit the model
model.fit(X, Y, validation_split=0.33, nb_epoch=150, batch_size=
10,
    callbacks=callbacks_list, verbose=0)
```

结果如下：

```
...
Epoch 00136: val_acc did not improve
Epoch 00137: val_acc did not improve
Epoch 00138: val_acc did not improve
Epoch 00139: val_acc did not improve
Epoch 00140: val_acc improved from 0.83465 to 0.83858, saving mo
del to weights.best.hdf5
Epoch 00141: val_acc did not improve
Epoch 00142: val_acc did not improve
Epoch 00143: val_acc did not improve
Epoch 00144: val_acc did not improve
Epoch 00145: val_acc did not improve
Epoch 00146: val_acc improved from 0.83858 to 0.84252, saving mo
del to weights.best.hdf5
Epoch 00147: val_acc did not improve
Epoch 00148: val_acc improved from 0.84252 to 0.84252, saving mo
del to weights.best.hdf5
Epoch 00149: val_acc did not improve
```

网络保存在：

```
weights.best.hdf5
```

14.4 导入保存的模型

保存点只保存权重，网络结构需要预先保存。参见第13章，代码如下：

```
# How to load and use weights from a checkpoint
from keras.models import Sequential
from keras.layers import Dense
from keras.callbacks import ModelCheckpoint
 import matplotlib.pyplot as plt
import numpy
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, init='uniform', activation='rel
u')) model.add(Dense(8, init='uniform', activation='relu'))
model.add(Dense(1, init='uniform', activation='sigmoid'))
# load weights
model.load_weights("weights.best.hdf5")
# Compile model (required to make predictions) model.compile(los
s='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
 print("Created model and loaded weights from file")
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter="
,")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# estimate accuracy on whole dataset using loaded weights
scores = model.evaluate(X, Y, verbose=0)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

结果如下：

```
Created model and loaded weights from file
acc: 77.73%
```

14.5 总结

本章关于在训练时保存检查点。总结一下：

- 如何在优化时保存网络
- 如何保存最好的网络
- 如何导入网络

14.5.1 下一章

本章关于建立保存点：下一章关于在训练时画性能图表。

第15章 模型训练效果可视化

查看训练效果的历史数据大有裨益。本章关于将模型的训练效果进行可视化。本章教你：

- 如何观察历史训练数据
- 如何在训练时绘制数据准确性图像
- 如何在训练时绘制损失图像

我们开始吧。

15.1 取历史数据

上一章说到Keras支持回调API，其中默认调用 `History` 函数，每轮训练收集损失和准确率，如果有测试集，也会收集测试集的数据。

历史数据会收集 `fit()` 函数的返回值，在 `history` 对象中。看一下到底收集了什么数据：

```
# list all data in history
print(history.history.keys())
```

如果是第7章的二分类问题：

```
['acc', 'loss', 'val_acc', 'val_loss']
```

可以用这些数据画折线图，直观看到：

- 模型收敛的速度（斜率）
- 模型是否已经收敛（稳定性）
- 模型是否过拟合（验证数据集）

以及更多。

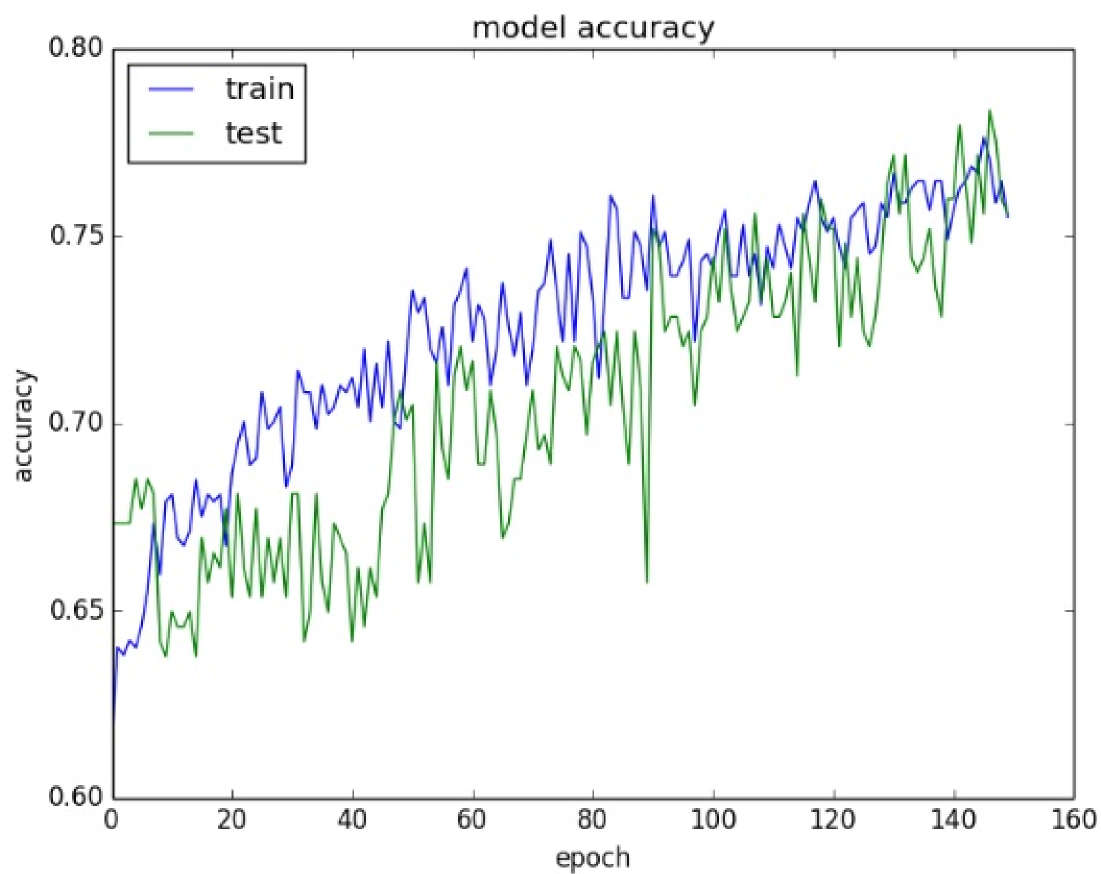
15.2 可视化Keras模型训练

收集一下第7章皮马人糖尿病模型的历史数据，绘制：

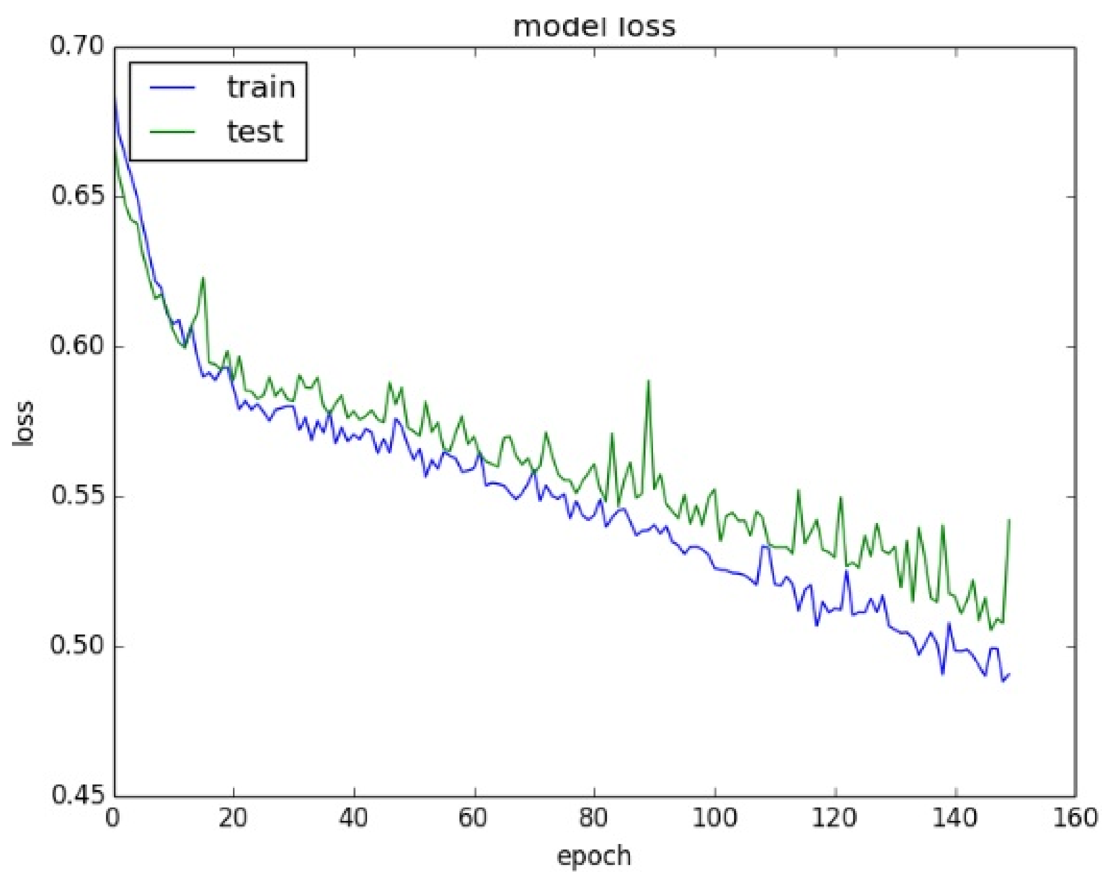
1. 训练和验证集的准确度
2. 训练和验证集的损失

```
# Visualize training history
from keras.models import Sequential
from keras.layers import Dense
import matplotlib.pyplot as plt
import numpy
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter="
,")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, init='uniform', activation='rel
u'))
model.add(Dense(8, init='uniform', activation='relu'))
model.add(Dense(1, init='uniform', activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metr
ics=['accuracy'])
# Fit the model
history = model.fit(X, Y, validation_split=0.33, nb_epoch=150, b
atch_size=10, verbose=0) # list all data in history
print(history.history.keys())
# summarize history for accuracy
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left') plt.show()
# summarize history for loss plt.plot(history.history['loss']) p
lt.plot(history.history['val_loss']) plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left') plt.show()
```

图像如下。最后几轮的准确率还在上升，有可能有点过度学习；但是两个数据集的效果差不多，应该没有过拟合。



从损失图像看，两个数据集的性能差不多。如果两条线开始分开，有可能应该提前终止训练。



15.3 总结

本章关于在训练时绘制图像。总结一下：

- 如何看历史对象
- 如何绘制历史性能
- 如何绘制两个数据集的不同性能

15.3.1 下一章

Dropout可以有效防止过拟合：下一章关于这个技术、如何在Keras中实现以及最佳实践。

第16章 使用Dropout正则化防止过拟合

Dropout虽然简单，但可以有效防止过拟合。本章关于如何在Keras中使用Dropout。本章包括：

- dropout的原理
- dropout的使用
- 在隐层上使用dropout

我们开始吧。

16.1 Dropout正则化

译者鄙校的Srivastava等大牛在2014年的论文《Dropout: A Simple Way to Prevent Neural Networks from Overfitting》提出了Dropout正则化。Dropout的意思是：每次训练时随机忽略一部分神经元，这些神经元dropped-out了。换句话讲，这些神经元在正向传播时对下游的启动影响被忽略，反向传播时也不会更新权重。

神经网络的所谓"学习"是指，让各个神经元的权重符合需要的特性。不同的神经元组合后可以分辨数据的某个特征。每个神经元的邻居会依赖邻居的行为组成的特征，如果过度依赖，就会造成过拟合。如果每次随机拿走一部分神经元，那么剩下的神经元就需要补上消失神经元的功能，整个网络变成很多独立网络（对同一问题的不同解决方法）的合集。

Dropout的效果是，网络对某个神经元的权重变化更不敏感，增加泛化能力，减少过拟合。

16.2 在Keras中使用Dropout正则化

Dropout就是每次训练按概率拿走一部分神经元，只在训练时使用。后面我们会研究其他的用法。

以下的例子是声呐数据集（第11章），用scikit-learn进行10折交叉检验，这样可以看出区别。输入变量有60个，输出1个，数据经过正则化。基线模型有2个隐层，第一个有60个神经元，第二个有30个。训练方法是随机梯度下降，学习率和动量较低。下面是基线模型的代码：

```
import numpy
import pandas
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.wrappers.scikit_learn import KerasClassifier
from keras.constraints import maxnorm
from keras.optimizers import SGD
from sklearn.cross_validation import cross_val_score
from sklearn.preprocessing import LabelEncoder
from sklearn.cross_validation import StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.grid_search import GridSearchCV
from sklearn.pipeline import Pipeline
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load dataset
dataframe = pandas.read_csv("sonar.csv", header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:60].astype(float)
Y = dataset[:,60]
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
encoded_Y = encoder.transform(Y)

# baseline
def create_baseline():
    # create model
    model = Sequential()
    model.add(Dense(60, input_dim=60, init='normal', activation=
'relu')) model.add(Dense(30, init='normal', activation='relu'))
model.add(Dense(1, init='normal', activation='sigmoid'))
    # Compile model
    sgd = SGD(lr=0.01, momentum=0.8, decay=0.0, nesterov=False)
    model.compile(loss='binary_crossentropy', optimizer=sgd, met
rics=['accuracy'])
    return model
numpy.random.seed(seed)
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(build_fn=create_baseli
ne, nb_epoch=300,
    batch_size=16, verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(y=encoded_Y, n_folds=10, shuffle=True, r
andom_state=seed)
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)
print("Accuracy: %.2f%% (%.2f%%)" % (results.mean()*100, results
.std()*100))
```

不使用Dropout的准确率是82%。

```
Accuracy: 82.68% (3.90%)
```

16.3 对输入层使用Dropout正则化

可以对表层用Dropout：这里我们对输入层（表层）和第一个隐层用Dropout，比例是20%，意思是每轮训练每5个输入随机去掉1个变量。

原论文推荐对每层的权重加限制，保证模不超过3：在定义全连接层时用 `W_constraint` 可以做到。学习率加10倍，动量加到0.9，原论文也如此推荐。对上面的模型进行修改：

```
# dropout in the input layer with weight constraint
def create_model1():
    # create model
    model = Sequential()
    model.add(Dropout(0.2, input_shape=(60,)))
    model.add(Dense(60, init='normal', activation='relu', W_cons
traint=maxnorm(3))) model.add(Dense(30, init='normal', activatio
n='relu', W_constraint=maxnorm(3))) model.add(Dense(1, init='nor
mal', activation='sigmoid'))
    # Compile model
    sgd = SGD(lr=0.1, momentum=0.9, decay=0.0, nesterov=False)
    model.compile(loss='binary_crossentropy', optimizer=sgd, met
rics=['accuracy'])
    return model
numpy.random.seed(seed)
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(build_fn=create_model1
, nb_epoch=300,
    batch_size=16, verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(y=encoded_Y, n_folds=10, shuffle=True, r
andom_state=seed)
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)
print("Accuracy: %.2f%% (%.2f%%)" % (results.mean()*100, results
.std()*100))
```

准确率提升到86%：

```
Accuracy: 86.04% (6.33%)
```

16.4 对隐层使用Dropout正则化

隐层当然也可以用Dropout。和上次一样，这次对两个隐层都做Dropout，概率还是20%：

```
# dropout in hidden layers with weight constraint
def create_model2():
    # create model
    model = Sequential()
    model.add(Dense(60, input_dim=60, init='normal', activation=
'relu',
W_constraint=maxnorm(3)))
    model.add(Dropout(0.2))
    model.add(Dense(30, init='normal', activation='relu', W_cons
traint=maxnorm(3)))
    model.add(Dropout(0.2))
    model.add(Dense(1, init='normal', activation='sigmoid'))
    # Compile model
    sgd = SGD(lr=0.1, momentum=0.9, decay=0.0, nesterov=False)
    model.compile(loss='binary_crossentropy', optimizer=sgd, met
rics=['accuracy'])
    return model
numpy.random.seed(seed)
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(build_fn=create_model2
, nb_epoch=300,
    batch_size=16, verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(y=encoded_Y, n_folds=10, shuffle=True, r
andom_state=seed)
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)
print("Accuracy: %.2f%% (%.2f%%)" % (results.mean()*100, results
.std()*100))
```

然而并没有什么卵用，效果更差了。有可能需要多训练一些吧。

```
Accuracy: 82.16% (6.16%)
```

16.5 使用Dropout正则化的技巧

原论文对很多标准机器学习问题做出了比较，并提出了下列建议：

1. Dropout概率不要太高，从20%开始，试到50%。太低的概率效果不好，太高有可能欠拟合。
2. 网络要大。更大的网络学习到不同方法的几率更大。
3. 每层都做Dropout，包括输入层。效果更好。
4. 学习率（带衰减的）和动量要大。直接对学习率乘10或100，动量设到0.9或0.99。
5. 限制每层的权重。学习率增大会造成权重增大，把每层的模限制到4或5的效果更好。

16.6 总结

本章关于使用Dropout正则化避免过拟合。总结一下：

- Dropout的工作原理是什么
- 如何使用Dropout
- Dropout的最佳实践是什么

16.6.1 下一章

在训练中调节学习率会提升性能。下一章会研究不同学习率的效果，以及如何在Keras中使用。

第17章 学习速度设计

神经网络的训练是很困难的优化问题。传统的随机梯度下降算法配合设计好的学习速度有时效果更好。本章包括：

- 调整学习速度的的原因
- 如何使用按时间变化的学习速度
- 如何使用按训练次数变化的学习速度

我们开始吧。

17.1 学习速度

随机梯度下降算法配合设计好的速度可以增强效果，减少训练时间：也叫学习速度退火或可变学习速度。其实就是慢慢调整学习速度，而传统的方法中学习速度不变。

最简单的调整方法是学习速度随时间下降，一开始做大的调整加速训练，后面慢慢微调性能。两个简单的方法：

- 根据训练轮数慢慢下降
- 到某个点下降到某个值

我们分别探讨一下。

17.2 电离层分类数据集

本章使用电离层二分类数据集，研究电离层中的自由电子。分类g（好）意味着电离层中有某个结构；b（坏）代表没有，信号通过了电离层。数据有34个属性，351个数据。

10折检验下最好的模型可以达到94~98%的准确度。数据在本书的data目录下，也可以自行下载，重命名为 `ionosphere.csv` 。数据集详情请参见UCI机器学习网站。

17.3 基于时间的学习速度调度

Keras内置了一个基于时间的学习速度调度器：Keras的随机梯度下降 `SGD` 类有 `decay` 参数，按下面的公式调整速度：

```
LearnRate = LearnRate x (1 / 1 + decay x epoch)
```

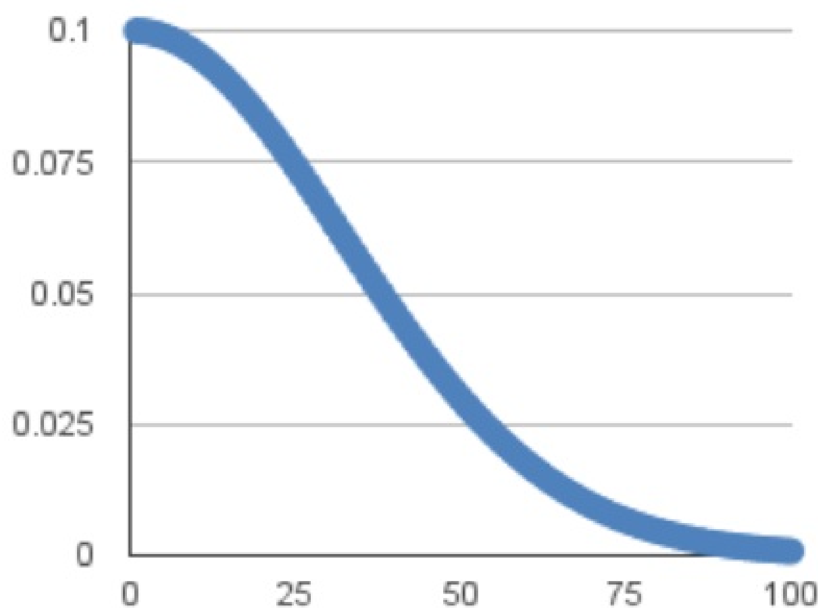
默认值是0：不起作用。

```
LearningRate = 0.1 * 1/(1 + 0.0 * 1)
LearningRate = 0.1
```

如果衰减率大于1，例如0.001，效果是：

```
Epoch Learning Rate
1 0.1
2 0.0999000999
3 0.0997006985
4 0.09940249103
5 0.09900646517
```

到100轮的图像：



可以这样设计：

```
Decay = LearningRate / Epochs
Decay = 0.1 / 100
Decay = 0.001
```

下面的代码按时间减少学习速度。神经网络有1个隐层，34个神经元，激活函数是整流函数。输出层是1个神经元，激活函数是S型函数，输出一个概率。学习率设到0.1，训练50轮，衰减率0.002，也就是0.1/50。学习速度调整一般配合动量使用：动量设成0.8。代码如下：

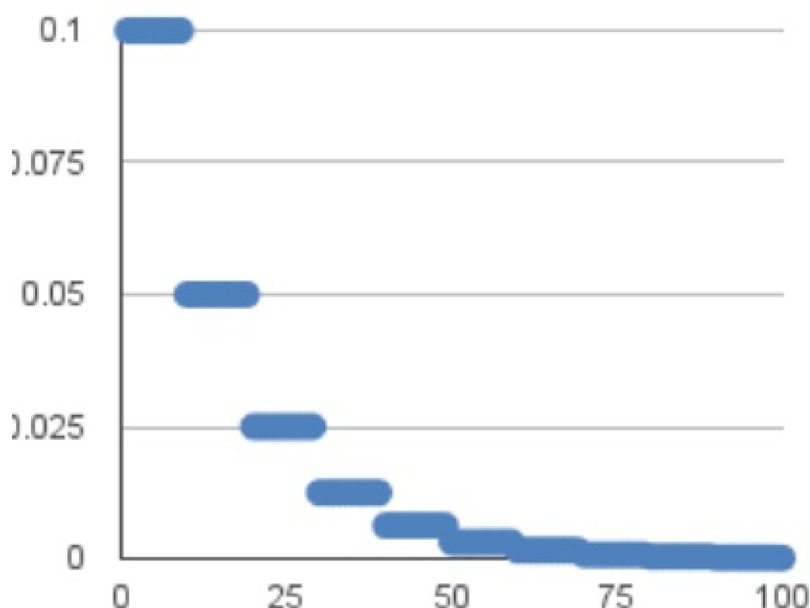
```
import pandas
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from sklearn.preprocessing import LabelEncoder
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load dataset
dataframe = pandas.read_csv("ionosphere.csv", header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:34].astype(float)
Y = dataset[:,34]
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
Y = encoder.transform(Y)
# create model
model = Sequential()
model.add(Dense(34, input_dim=34, init='normal', activation='rel
u')) model.add(Dense(1, init='normal', activation='sigmoid'))
# Compile model
epochs = 50
learning_rate = 0.1
decay_rate = learning_rate / epochs
momentum = 0.8
sgd = SGD(lr=learning_rate, momentum=momentum, decay=decay_rate,
 nesterov=False) model.compile(loss='binary_crossentropy', optim
izer=sgd, metrics=['accuracy'])
# Fit the model
model.fit(X, Y, validation_split=0.33, nb_epoch=epochs, batch_si
ze=28)
```

训练67%，测试33%的数据，准确度达到了99.14%，高于不使用任何优化的95.69%：

```
235/235 [==============================] - 0s - loss: 0.0607 - a
cc: 0.9830 - val_loss:
    0.0732 - val_acc: 0.9914
Epoch 46/50
235/235 [==============================] - 0s - loss: 0.0570 - a
cc: 0.9830 - val_loss:
    0.0867 - val_acc: 0.9914
Epoch 47/50
235/235 [==============================] - 0s - loss: 0.0584 - a
cc: 0.9830 - val_loss:
    0.0808 - val_acc: 0.9914
Epoch 48/50
235/235 [==============================] - 0s - loss: 0.0610 - a
cc: 0.9872 - val_loss:
    0.0653 - val_acc: 0.9828
Epoch 49/50
235/235 [==============================] - 0s - loss: 0.0591 - a
cc: 0.9830 - val_loss:
    0.0821 - val_acc: 0.9914
Epoch 50/50
235/235 [==============================] - 0s - loss: 0.0598 - a
cc: 0.9872 - val_loss:
    0.0739 - val_acc: 0.9914
```

17.3 基于轮数的学习速度调度

也可以固定调度：到某个轮数就用某个速度，每次的速度是上次的一半。例如，初始速度0.1，每10轮降低一半。画图就是：



Keras的 `LearningRateScheduler` 作为回调参数可以控制学习速度，取当前的轮数，返回应有的速度。还是刚才的网络，加入一个 `step_decay` 函数，生成如下学习率：

```
LearnRate = InitialLearningRate x Droprate ^ floor((1+Epoch)/Epo
chDrop)
```

InitialLearningRate是初始的速度，DropRate是减速频率，EpochDrop是降低多少：

```
import pandas
import pandas
import numpy
import math
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from sklearn.preprocessing import LabelEncoder
from keras.callbacks import LearningRateScheduler
# learning rate schedule
def step_decay(epoch):
  initial_lrate = 0.1
  drop = 0.5
  epochs_drop = 10.0
  lrate = initial_lrate * math.pow(drop, math.floor((1+epoch)/ep
ochs_drop))
  return lrate
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load dataset
dataframe = pandas.read_csv("../data/ionosphere.csv", header=None
)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:34].astype(float)
Y = dataset[:,34]
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
Y = encoder.transform(Y)
# create model
model = Sequential()
model.add(Dense(34, input_dim=34, init='normal', activation='rel
u'))
model.add(Dense(1, init='normal', activation='sigmoid'))
# Compile model
sgd = SGD(lr=0.0, momentum=0.9, decay=0.0, nesterov=False) model
.compile(loss='binary_crossentropy', optimizer=sgd, metrics=['ac
curacy'])
# learning schedule callback
lrate = LearningRateScheduler(step_decay)
callbacks_list = [lrate]
# Fit the model
model.fit(X, Y, validation_split=0.33, nb_epoch=50, batch_size=28
, callbacks=callbacks_list)
```

效果也是99.14%，比什么都不做好：

```
Epoch 45/50
235/235 [==============================] - 0s - loss: 0.0546 - a
cc: 0.9830 - val_loss:
    0.0705 - val_acc: 0.9914
Epoch 46/50
235/235 [==============================] - 0s - loss: 0.0542 - a
cc: 0.9830 - val_loss:
    0.0676 - val_acc: 0.9914
Epoch 47/50
235/235 [==============================] - 0s - loss: 0.0538 - a
cc: 0.9830 - val_loss:
    0.0668 - val_acc: 0.9914
Epoch 48/50
235/235 [==============================] - 0s - loss: 0.0539 - a
cc: 0.9830 - val_loss:
    0.0708 - val_acc: 0.9914
Epoch 49/50
235/235 [==============================] - 0s - loss: 0.0539 - a
cc: 0.9830 - val_loss:
    0.0674 - val_acc: 0.9914
Epoch 50/50
235/235 [==============================] - 0s - loss: 0.0531 - a
cc: 0.9830 - val_loss:
    0.0694 - val_acc: 0.9914
```

17.5 调整学习速度的技巧

这些技巧可以帮助调参：

1. 增加初始学习速度。因为速度后面会降低，一开始速度快点可以加速收敛。
2. 动量要大。这样后期学习速度下降时如果方向一样，还可以继续收敛。
3. 多试验。这个问题没有定论，需要多尝试。也试试指数下降和什么都不做。

17.6 总结

本章关于调整学习速度。总结一下：

- 调整学习速度为什么有效
- 如何在Keras使用基于时间的学习速度下降
- 如何自己编写下降速度函数

17.6.1 下一章

第四章到此结束，包括Keras的一些高级函数和调参的高级方法。下一章研究卷积神经网络（CNN），在图片和自然语言处理上尤为有效。

第五部分 卷积神经网络